Responsive Execution of Parallel Programs in Distributed Computing Environments

D I S S E R T A T I O N

zur Erlangung des akademischen Grades
doctor rerum naturalium
(dr. rer. nat.)
im Fach Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät II
Humboldt-Universität zu Berlin

von
Herrn Dipl.-Inform. Fritz Holger Karl
geborem am 15.2.1970 in Eberbach

Präsident der Humboldt-Universität zu Berlin:
Prof. Dr. sc. Bodo Krause

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät II:
Prof. Dr. sc. Heinz Müller

Gutachter:

1. Prof. Dr. Miroslaw Malek
2. Prof. Dr. Stefan Jähnichen
3. Prof. Dr. Zvi M. Kedem

eingereicht am: 11. Juni 1999
Tag der mündlichen Prüfung: 3. Dezember 1999

Zusammenfassung

Vernetzte Standardarbeitsplatzrechner (sog. Cluster) sind eine attraktive Umgebung zur Ausführung paralleler Programme; für einige Anwendungsgebiete bestehen jedoch noch immer ungelöste Probleme. Ein solches Problem ist die Verläßlichkeit und Rechtzeitigkeit der Programmausführung: In vielen Anwendungen ist es wichtig, sich auf die rechtzeitige Fertigstellung eines Programms verlassen zu können. Mechanismen zur Kombination dieser Eigenschaften für parallele Programme in verteilten Rechenumgebungen sind das Hauptanliegen dieser Arbeit.

Zur Behandlung dieses Anliegens ist eine gemeinsame Metrik für Verläßlichkeit und Rechtzeitigkeit notwendig. Eine solche Metrik ist die Responsivität, die für die Bedürfnisse dieser Arbeit verfeinert wird. Als Fallstudie werden Calypso und Charlotte, zwei Systeme zur parallelen Programmierung, im Hinblick auf Responsivität untersucht und auf mehreren Abstraktionsebenen werden Ansatzpunkte zur Verbesserung ihrer Responsivität identifiziert. Lösungen für diese Ansatzpunkte werden zu allgemeineren Mechanismen für (parallele) responsive Dienste erweitert.

Im Einzelnen handelt es sich um 1. eine Analyse der Responsivität von Calypsos “eager scheduling” (ein Verfahren zur Lastbalancierung und Fehlermaskierung), 2. die Behebung eines “single point of failure,” zum einen durch eine Responsivitätsanalyse von Checkpointing, zum anderen durch ein auf Standardschnittstellen basierendes System zur Replikation bestehender Software, 3. ein Verfahren zur garantierten Ressourcenzuteilung für parallele Programme und 4. die Einbeziehung semantischer Information über das Kommunikationsmuster eines Programms in dessen Ausführung zur Verbesserung der Leistungsfähigkeit. Die vorgeschlagenen Mechanismen sind kombinierbar und für den Einsatz in Standardsystemen geeignet. Analyse und Experimente zeigen, daß diese Mechanismen die Responsivität passender Anwendungen verbessern.

Schlagwörter:
paralleles und verteiltes Rechnen, Fehlertoleranz, Echtzeit, Responsivitä

Abstract

Clusters of standard workstations have been shown to be an attractive environment for parallel computing. However, there remain unsolved problems to make them suitable to some application scenarios. One of these problems is a dependable and timely program execution: There are many applications in which a program should be successfully completed at a predictable point of time. Mechanisms to combine the properties of both dependable and timely execution of parallel programs in distributed computing environments are the main objective of this dissertation.

Addressing these properties requires a joint metric for dependability and timeliness. Responsiveness is such a metric; it is refined for the purposes of this work. As a case study, Calypso and Charlotte, two parallel programming systems, are analyzed and their shortcomings on several abstraction levels with regard to responsiveness are identified. Solutions for them are presented and generalized, resulting in widely applicable mechanisms for (parallel) responsive services.

Specifically, these solutions are: 1) a responsiveness analysis of Calypso's eager scheduling (a mechanism for load balancing and fault masking), 2) ameliorating a single point of failure by a responsiveness analysis of checkpointing and by a standard interface-based system for replication of legacy software, 3) managing resources in a way suitable for parallel programs, and 4) using semantical information about the communication pattern of a program to improve its performance. All proposed mechanisms can be combined and are suitable for use in standard environments. It is shown by analysis and experiments that these mechanisms improve the responsiveness of eligible applications.

Keywords:
parallel and distributed computing, fault tolerance, real time, responsiveness

To Jürgen

Acknowledgments

I would like to express my sincere thanks to a number of people for making it possible for me to complete this dissertation.

My advisor Prof. Dr. Miroslaw Malek of Humboldt University has provided an interesting research topic and a fruitful atmosphere. He has always encouraged critical discussion and shown great patience and encouragement, particularly in difficult periods. He has certainly helped shaping this dissertation to a large extend. I am very grateful to him.

My deep gratitude also goes to my second advisor, Prof. Dr. Zvi Kedem of New York University for inviting me to spend eight months with his research group at NYU. He and his group were very welcoming and showed immense hospitality. Dr. Kedem provided me with every conceivable support and I learned a lot from him about research. I am deeply indebted to him.

I am also grateful to a large number of colleagues for collaboration, discussions, joint work and research, and the occasional musings about the meaning of everything. I can name but a few, yet I salute them all. These colleagues include Fangzhe Chang, Dr. Gerhard Fohler, Peter Ibach, Dr. Mehmet Karaul, Lars Küttner, Dr. Andreas Polze, Jan Richling, Birgit Schiefner, Janek Schwarz, Dr. Peter Wyckhoff, and Yuanyuan Zhao, yet I would like to particularly mention Dr. Matthias Werner and Dr. Arash Baratloo: with both I have shared a productive, enjoyable working relationship which I cannot appreciate enough.

Thanks are also due to the graduate program “Kommunikationsbasierte Systeme” of the DFG, and in particular to its speaker, Prof. Dr. G. Hommel, for providing financial support in a challenging scientific context. Also, Frau Sabine Becker offered help, patience, and good humor in all administrative matters.

And last but not least, my deepest, heartfelt thanks go to my friend Jürgen for his support, patience, love, and encouragement during sometimes difficult times.

Contents

List of Figures

List of Tables

Abbreviations and Acronyms

API Application Programming Interface

ATM Asynchronous Transfer Mode

BSP Bulk Synchronous Parallel

COM Component Object Model

CORBA Common Object Request Broker Architecture

CORE Consensus for Responsiveness

COTS Commercial Off-The-Shelf.

CPU Central Processing Unity.

CRC Cyclic Redundancy Check

CRCW Concurrent Read Concurrent Write

CREW Concurrent Read Exclusive Write

CRL C Region Library

CSP Communicating Sequential Processes

CT Consensus Timeout

DAG Directed Acyclic Graph

DCOM Distributed Common Object Model

DMA Direct Memory Access

DSM Distributed Shared Memory

Gbps Gigabits per second

EDF Earliest Deadline First

ENIAC Electronic Numerical Integrator and Computer

ERICA Error Resistant Interactively Consistent Architecture

FTDIO Fault-Tolerant Distributed I/O

GCL Group Communication Layer

HTML Hypertext Markup Language

HTTP Hypertext Transfer Procotol

HU Humboldt University

IBM International Business Machines

I/O Input/Output

IP Internet Protocol

ITU International Telecommunication Union

JDK Java Development Kit

JIT Just-In-Time

JT Join Timeout

Kbps Kilobits per second

LAN Local Area Network

MAFT Multicomputer Architecture for Fault Tolerance

Mars Maintainable real-time system

Mbps Megabits per second

Milan Metacomputing in large asynchronous networks

MPI Message Passing Interface

MPI/RT Message Passing Interface real-time extension

MTBF Mean Time Between Failures

NCAPS NonStop Cluster Application Protection System

NYU New York University

PC Personal Computer

PCI Peripheral Component Interconnect

PERC Portable Executive for Reliable Control

POSIX Portable Operating System Interface

PPM Process Pair Manager

PVM Parallel Virtual Machine

PRAM Parallel Random Access Machine

QoS Quality of Service

RMI Remote Method Invocation

RM-ODP Reference Model for Open Distributed Processing

RMS Rate Monotonic Scheduling

rv random variable

RSVP Resource Reservation Protocol

SAN System Area Network

SCI Scalable Coherent Interface

SI International System of Units

SIFT Software Implemented Fault Tolerance

SMP Symmetric Multiprocessing

SONiC Shared Objects Network-interconnected Computer

TCP Transmission Control Protocol

TDMA Time-Division Multiple Access

TFT Transparent Fault Tolerance

TIES Two-phase Idempotent Execution Strategy

TLB Translation Look-aside Buffer

TLT Token Loss Timeout

TRT Token Retransmission Timeout

TTP Time-Triggered Protocol

UDP User Datagram Protocol

URL Uniform Resource Locator

VIA Virtual Interface Architecture

VLSI Very Large Scale Integration

WAN Wide Area Network

WWW World Wide Web

Some other general notation:

$\mathbb{R}$ is the set of real numbers.

$\mathbb{R}^+$ is the set of positive real numbers.

$\mathbb{R}_0^+$ is the set of positive real numbers including 0.

$\mathbb{N}$ is the set of natural numbers (including 0).

D^n is, for any set D, the n-fold Cartesian product of D.

$F_X(x)$ is the cumlative distribution for a random variable X: $\Pr(X \leq x) = F_X(x)$.

$f_X(x)$ is the probabilistic density function for a random variable X:
$f_X(x) = \lim_{\Delta t \to 0} \frac{\Pr(X<x+\Delta t)-\Pr(X<x)}{\Delta t}$.

One of the symptoms of an approaching nervous breakdown is the belief that one's work is terribly important.
– Bertrand Russell

Chapter 1

Introduction

Clusters of workstations are a viable alternative to custom-designed supercomputers for many applications, yet a number of problems remain to be solved before clusters are a superior choice. In this chapter, dependable and timely execution of parallel programs on a cluster is identified as one of these problems and a brief outline how this dissertation proposes to approach this problem is given.

1.1 Are clusters competitive?

Solving problems that require a very large amount of computational resources is a traditional problem of computer science. Examples for such problems are numerous and start with the computing of artillery firing tables on world's first general-purpose Electronic Numerical Integrator and Computer (ENIAC) in the mid-1940's, and extend to today's grand challenge problems like climate and weather prediction, the simulation of aging processes in nuclear warheads, or the search for oil deposits.

The classical response to this need was and to a large extent still is large, special-purpose computers: vector computers (e.g., the Cray I) or today more and more massively parallel machines (e.g., the Connection Machine [106], the *T [213] or the Alewife [3]). In such parallel systems, many processing units work in concert to provide a larger computational power than any single machine could. Typically, these machines are engineered to embody the best possible technology and, consequently, are very expensive. But they do provide exceptional performance for a large variety of problems.

At the other end of the spectrum, the advent of Personal Computers (PC) and workstations, facilitated by progress in Very Large Scale Integration (VLSI) and microprocessor technology, has revolutionized computer science. The performance of such machines has surpassed that of early generation supercomputers, and consequently, many of today's supercomputers indeed are based on such standard components. Moreover, leveraging Gordon Bell's law that promises a 10% cost reduction for every doubling in volume, personal computers and workstations now have an unsurpassed price/performance ratio. Such workstations are cheap, almost ubiquitous, and sometimes their use is even free if the idle times of already existing machines can be exploited.

The increased usage of workstations has been accompanied by the need to interconnect them in local area and wide area installations, eventually resulting into a pervasiveness of interconnected machines and the Internet. This has lead to much improved, and much cheaper, networking technologies that deliver high bandwidth and low latencies at an affordable price.

With both computers and networks being widely available at low cost, many research projects have suggested to use such connected clusters of workstations for parallel computing (pioneered by projects like Parallel Virtual Machine (PVM) [279] and others), rivalling the traditional supercomputer architectures. Such clusters have a number of appealing aspects. Since they are made of standard, off-the-shelf components, they are able to track the technological progress much more closely than special-purpose architectures that

suffer from a long development cycle, resulting in a very short time to market for cluster-based systems. Additionally, they can leverage the benefits of mass economics much more easily since the development cost can be spread over a much larger number of users. Based on such arguments, ANDERSON et al. [9] make a compelling case for these "networks of workstations."

Therefore, two main approaches to building high-performance systems can be identified. On the one hand, special-purpose supercomputers, constructed out of standard components, but with much additional and custom-designed hardware and software for composing these components into a single machine (indeed, completely custom-designed machines like the Connection Machine are no longer viable for the market); on the other hand, clusters of standard Commercial Off-The-Shelf (COTS) PCs or workstations, interconnected by standard networking technology.[1] The term "cluster" is still not quite clearly defined[2]; one possible definition is "A cluster is a type of parallel or distributed system that consists of a collection of interconnected whole computers used as a single, unified computing resource." [223, p. 72], where "whole computer" typically refers to a normal computer system that can be used on its own (including CPU, memory, I/O, operating system). In the context of this dissertation, a cluster is not necessarily used as a single resource, but the constituent machines might be shared with other, possibly interactive, users.

The main argument for clusters is a far superior cost/performance ratio while being able to deliver competitive performance when compared to supercomputers. As an example, price and performance of a typical cluster of COTS machines (consisting of Dell Optiplex GX1p machines with a Myrinet interconnection network) are compared in Table 1.1 with a massively parallel supercomputer (an SGI Origin 2000 [164]).[3]

	COTS PC	Supercomputer
Performance per node	Intel Pentium III (500 MHz, with 15.9 SpecInt95, 21.7 SpecFp95, 128 MB RAM)	MIPS R10000 (250 MHz, with 14.7 SpecInt95, 24.5 SpecFp95, 128 MB RAM)
Network	Myrinet (1 μs latency, 1.28 Gbps bandwidth)	Custom-build (0.5 μs latency, 2.56 Gbps bandwidth)
Cost per node	\$ 4,100	\$ 16,000

Table 1.1: Cost/performance comparison of COTS PC and supercomputer in a 32-node configuration (as of May 1999).

Evidently the PC cluster has comparable performance—with network performance about a factor of two lower—at a fraction of the cost of the supercomputer. The possible advantages of supercomputers are not sufficient to make up for their disadvantages, which has led to direct consequences: A number of producers of parallel machines (e.g., Thinking Machines) have filed for bankruptcy [84], others have reoriented themselves

[1] The term "commercial off-the-shelf" is somewhat problematic. Usually this term is meant to refer to systems that can be bought anywhere and are typical examples of the current technology for desktop machines. It is important to make this restriction, since even a very expensive supercomputer can be "off-the-shelf" of its manufacturer. Where is the border line? Does a machine like a Sun Enterprise Server with 64 processors still count as COTS? Strictly speaking, yes—it can be ordered from Sun without any further ado. But that is not the usual connotation of this term. The adherence to industry standards is also usually implied by the notion of COTS—although for all practical matters, this industry standard is dictated by one or two companies. Nevertheless, this term is be used here with the assumption that it is clear by the context in which it is used both in this dissertation and in today's computer science discussion.

[2] As witnessed by the lively discussion about this issue in the mailing list of the IEEE Task Force on Cluster Computing [110].

[3] Prices are as of May 1999, obtained from the World Wide Web (WWW) pages of Dell and Myricom and from a local sales representative of SGI; the Myrinet configuration follows suggestions in [57]. For both configurations, 32 nodes are assumed. Performance numbers are obtained from the WWW pages of Intel and SGI as well as from [267].

towards fault-tolerant computing or transaction processing—the market share for supercomputers remains at about 3%. Some supercomputer designs are partially based on standard workstations, but enhanced with special-purpose interconnection networks; the IBM SP-2 is a good example for a machine of this type.

All these factors contribute to making clusters a very viable alternative to custom-designed supercomputers. Consequently, there is an already large and growing interest in industry, not only with regard to parallel systems. As an example for this trend, consider Microsoft's Windows/NT cluster system, Wolfpack [257], or the Virtual Interface Architecture (VIA) proposal [291], jointly promoted by Intel, Microsoft and Compaq. VIA describes an architecture for the interface between computer systems and high-performance networks which aims at reducing application-level latency.

1.2 Problems with clusters

Given all these advantages of clusters like superior price/performance and time to market, why are supercomputers still manufactured and sold? Apparently, there are still some areas where clusters do not constitute an acceptable solution. This section gives an overview of such issues and identifies areas that require additional research efforts.

1.2.1 Communication

The most evident problem of clusters—compared to supercomputers—is the efficiency of distributed computations. Since the CPU performance available in COTS systems is comparable and, owing to the long time-to-market of custom designs, sometimes even superior to that in custom-built supercomputers (as has been indicated by Table 1.1), the communication performance characterized by bandwidth, latency and overhead is the determining factor for parallel performance. This in turn depends mostly on the communication hardware and the integration of communication into the endsystem.

A number of challenges make high communication performance more difficult to achieve in a COTS cluster than in a supercomputer. The most important ones are: physical distance between nodes, integration of the network interface in a node's hardware/software architecture, and the need for a higher level of protection of resources.

The small physical distances between nodes in a supercomputer allow the use of faster and more reliable communication hardware than in a cluster. The lower reliability of Local Area Networks (LAN) has forced clusters to use heavy-weight protocol stacks like Transmission Control Protocol (TCP)/Internet Protocol (IP), incurring a high performance penalty. This shortcoming is rapidly remedied with the advent of what has been called System Area Networks (SAN) [109]: Myricom's Myrinet [39] or Compaq's Servnet [252] are examples for networks that deliver Gigabits per second (Gbps) bandwidth and latencies of tens of nanoseconds, with very high reliability.

The second problem is integration of the network interface into the host architecture. Typically, network interfaces are connected to the I/O system of a COTS machine, whereas in a supercomputer, the network interface can be connected directly to the memory bus or the processor itself. This incurs performance penalties, but has been addressed by much research (an overview can be found, e.g., in [205]).

The question of virtualizing the network interface and protecting it from conflicting accesses from several processes constitutes the third problem. Since a supercomputer is often used by only one application at a time, this application can be granted uncontrolled access to a system resource like the network interface. In a COTS machine, on the other hand, the network interface has to be designed to protect multiple applications, which share a single machine, from each other; e.g., an application must not be allowed to receive messages addressed to another application.

Closely related to the question of communication performance is the question of synchronization. Synchronization is, in a certain sense, a prerequisite for communication, and some programming models make this very explicit. Additionally, closely synchronized execution of distributed parts of the program can have a large impact on performance. This is discussed in more detail in Chapter 8.

While the communication performance of clusters is, owing to these problems, not yet quite as high as that of supercomputers, much progress has been made (a more detailed discussion can be found in Section 3.2). And with communication performance, the performance delivered to a parallel application also increases. Pure performance is therefore not the issue of this dissertation.

1.2.2 Programming models

Writing a parallel program to execute in a cluster environment is a complicated endeavor compared to a supercomputer system. The machines in a cluster can well be heterogenous or at least of varied speed. Failures of machines may occur more likely in a cluster than in a closely administered machine, in particular if the machines in a cluster are shared with interactive users. The number of available machines in a cluster can well vary between different invocations of the same program. And although high-performance communication interfaces are becoming available for clusters, they are usually not nearly as well integrated in a cluster's operating systems as are their counterparts in parallel supercomputers.

Other issues have more to do with programmability and appear in both supercomputers and clusters: e.g., distributing complex data structures over connected machines. Such questions often have comparatively simple solutions in supercomputers since their tighter integration of computation and communication allows more convenient programming models such as Distributed Shared Memory (DSM).

This observation is key to many approaches: programming models with a higher level of abstraction hide irrelevant details from a programmer and allow him to concentrate on application-specific problems. It is therefore promising to hide cluster-specific complexities behind a simple programming model as well. The systems of Metacomputing in large asynchronous networks (Milan) project [23, 27, 64] follow this approach to hide complexities such as number, different speeds, and faults of machines by separating the semantics of a program from environment-specific issues. Calypso, one of these systems, is be described in more detail in Chapter 4.

Additionally, such abstract programming models lend themselves naturally to extending their semantics for inclusion of new properties. It is conceptually easy just to add yet another hidden complexity to such a model; nonetheless, the programmer and/or user have to provide sufficient information to make this possible. A mechanism for a programmer to express additional information about a program is introduced in Chapter 9.

For users of high-performance systems, the abstraction level offered by such programming models is often still too low-level. A number of projects target tools, libraries and runtime environments that provide easier adaption of numerical problems, as well as interaction and integration of existing applications. Tradeoffs between performance and usability, however, are still an open question. A recent description of some such projects can be found in [244]

1.2.3 Intrusiveness

Intimately tied with the idea of COTS systems is the notion of non-intrusiveness: Not only should readily available components be used in system construction; moreover, they should be used as is, without requiring any unnecessary modifications. This idea is in sharp contrast with the design of supercomputers. While they increasingly often use standard components like microprocessors, they are often modified or endowed with additional, non-standard, custom-specific hardware (like interconnection networks, buses, cache controllers, or even such low-level components as the Translation Look-aside Buffer (TLB)) or software (in particular, modified operating systems).

For a truly COTS-based system, such intrusions are unacceptable. Any add-ons or modifications must always ensure the correct function of all services the system offered before and must coexist without interference with these standard services—programs should still run, machines perform their functions as before, interfaces must not be changed. Also, no knowledge about internal mechanisms should be exploited, if it is available at all.

Such non-intrusiveness has implications for the design of additional functionalities. In particular, middleware approaches that are layered on top of existing services without blocking access to lower layers are good

candidates. In such an approach, an existing system is enhanced with additional software (and, if necessary, hardware) that provides the necessary functionality on top of the original system interfaces, without modifying them, but only adding new functionality to it—nothing that need not be modified should be modified. Any add-ons must be strictly transparent.

Similarly, the only acceptable interfaces for a middleware solution are those that are provided by the system in a standard manner. A middleware that adds new properties should adhere to all possible conventions of program interoperability. While this limits the space of potential solutions, it is a sine qua non of any COTS approach.

1.2.4 Management

A potential shortcoming of clusters is the lack of central information about the state of the cluster as a whole. In a supercomputer, there is typically some centralized instance that provides a single representation of the entire system. This facilitates questions of administration, sharing of resources among multiple jobs (e.g., in a space-sharing fashion), fault masking (e.g., not allocating jobs to a failed processor) or timely coordination of resource usage (e.g., coscheduling [219]) and other system and resource management issues.

While it is possible to provide such a single image of the state of a cluster, it is an expensive undertaking in terms of runtime overhead and might nonetheless result in information of only limited precision. It is therefore a legitimate question to ask how to decentralize these problems and how to solve them in a less tightly-coupled environment such as a cluster. In [188], albeit in a slightly different context, three possible approaches to such a question are discussed. The "omniscient" approach corresponds to the centralized information as found in a supercomputer. Obvious problems with this approach include scalability and fault tolerance. An alternative is "tamed nondeterminism", implemented via consensus protocols, which means the periodic exchange of knowledge and the achievement of consensus on future actions. Third, completely independent systems pursue their own objectives in an autonomous fashion.

These question become particularly interesting when combined with the demand for non-intrusive solutions. Also, management is never an end in itself but only a means for other objectives. As a concrete cases of the issues arising in system management, managing resources in a cluster-based system so as to guarantee access to resources for both sequential and parallel programs is discussed in Chapter 8.

1.2.5 Predictability and timeliness

In a typical supercomputer environment, users of such a machine have yet another requirement: they want to depend on their programs being completed at a certain time. Historically, this has been more of an obligation to users because maximum runtimes were and are often used to plan the order of program execution to maximize the utilization of a supercomputer. Over time, this has developed more into an expectation and people are often willing to bear the inherent burdens (like specifying maximal resource requirements of a program when submitting a program) to be able to rely on such predictable completions.

Such an ability to complete programs in time is crucial in a number of applications. Examples include signal processing in real time (e.g., processing radar signals [193]), weather-related services (LEE et al. [166] describe a scenario where an IBM SP-2 has been used as part of a wide area scenario to process satellite images for cloud detection in nearly real time), the "almost real time" visualization of microtomography experiments [296], or even large-scale battlefield simulations (where interactiveness makes timely completion of programs an indispensable condition). Therefore, executing programs in a timely manner is a capability that clusters should also be able to provide.

Meeting this requirement of predictable and timely execution of programs is not a simple task in a cluster. A number of factors contribute to this difficulty. One is the fact that clusters are often used in a time-shared fashion. This sharing can happen among multiple parallel programs or between parallel programs and interactive users. In either case, there is contention for resources, possibly limiting predictability and timeliness if this contention in itself is unpredictable. This contention raises the need for resource management functionality to deal with it.

A second factor is related to this time-shared usage: clusters are commonly less well guarded then supercomputers; it is, e.g., readily possible that someone reboots a machine within a cluster. Such rebooting has similar consequences as a crash fault of a machine, and faults in general are always a possibility that must be dealt with. The existence of faults also implies that, while predictability can be a useful tool to achieve timeliness, it is not a sufficient property: A program that always crashes before producing any results is perfectly predictable (and might even crash on time), but useless. Consequently, timeliness must be accompanied by dependability and corresponding fault-tolerance mechanisms to be useful.

The third factor is that, even given information about the program, and even in an absence of faults, the particular execution regime of a parallel programming system can introduce some uncertainty over the runtime of a program (e.g., owing to random effects like caching during program execution). This uncertainty is aggravated by faults and requires an analysis of the program runtimes in an appropriate model. Similarly, the technical infrastructure of a typical cluster may not be as suitable to timely execution of parallel programs as that of a supercomputer, potentially owing to rather low-level properties: the inherently probabilistic Ethernet is less predictable than a deterministic interconnection networks.

These factors show that, while timely program execution is necessary for a growing number of applications, there are still many open questions to be solved before a cluster of workstations is a suitable environment for such applications. This dissertation attempts to contribute a few solutions to some aspects of this problem.

1.3 Problem Definition

In the previous Section 1.2, some problem areas have been pointed out where clusters of workstations are still in need of improvements. The last one, timeliness accompanied by dependability, is particularly crucial. This dissertation therefore focuses on the feasibility of making the execution of parallel programs timely and dependable, mostly in clusters of workstations, but also with the perspective of wide area computing.

More precisely: What mechanisms, paradigms, analyses, or implementation techniques are needed to execute a parallel program on a set of independent, off-the-shelf machines so that it is possible to make some kind of assurances about the time needed to execute this program—determining the proper nature of these assurances is by itself part of the problem—and how can these results be applied towards improving these assurances.

Devising such assurances about the runtime of a program is complicated by a number of factors. One is the program itself and proper assumptions about the program as well as its execution environment. Another such factor are faults in the execution environment. A third factor is the presence of load on the cluster machines. These problems have to be addressed.

Targeting clusters of commodity, off-the-shelf systems limits the range of possible solutions: All mechanisms must be compatible with standard systems environments (such as hardware or operating system) as found in today's typical workstation and PC architectures; incompatible mechanisms would not qualify as valid solutions. Ideally, the problem definition therefore demands solutions that intrude as little as possible into a given system.

It should also be pointed out that mere performance for parallel programs is not the objective of this dissertation. As indicated in Section 1.2.1 and as is discussed in detail in Section 3.2, many research projects have considered the question of high-performance computing in clusters of workstations, and much progress has already been made. Therefore, addressing the need for timeliness and dependability seems more pressing; any solution should nevertheless be competitive with respect to performance.

1.4 Outline

The first problem to solve is a precise definition of predictability, timeliness, and dependability in the context of (parallel) program execution. The intuitive concept of predictability serves as a starting point in Chapter 2 and it is discussed in the context of the notion of Quality of Service. From this discussion, the concept of responsiveness emerges as a (in this context) suitable formalization of the intuition. Responsiveness allows a

succinct characterization of the probabilistic behavior of a service—here, the execution of a program—in real time in the presence of faults and is used in this dissertation as a joint metric for timeliness and dependability.

The necessity of employing middleware solutions has been argued in Section 1.2.3. It therefore appears wise to concentrate on a specific system and to investigate paradigmatically in how far middleware solutions can contribute to the desired goal of increased responsiveness. An overview of what systems are currently available to serve as a starting point for this endeavor is given in Chapter 3 and the systems developed in the Milan project at the New York University are selected as a case study. Calypso [23], one of Milan's systems, is then analyzed with regard to responsiveness in Chapter 4 and four main areas for improvements are identified: The need for an analysis of its execution strategy, ameliorating the problems caused by a single point of failure, managing resources in a way suitable for parallel programs, and limiting the communication overhead.

An answer to the first of these problems is provided in Chapter 5: An analysis of Calypso's so-called eager scheduling execution strategy. Eager scheduling is a generally usable scheduling mechanism that integrates fault masking and load balancing. The execution time of a program when this scheduling algorithm is used is analyzed for heterogenous, potentially failing machines and two different sets of assumptions about the executed tasks.

The problem of a single point of failure in Calypso is then considered in Chapter 6 and it is investigated how checkpointing can be used to solve this problem. In particular, since responsiveness is the main objective, a novel analysis of the checkpointing interval problem is presented, maximizing the responsiveness of a service with checkpointing. This theoretical analysis is then additionally exemplified by experiments with a Calypso version extended by checkpointing functionality.

The problem of a single point of failure is reconsidered in Chapter 7 under a different perspective: replication. Replication is a widely-used technique to improve the fault tolerance of many systems. In practical settings, however, the coordination of replicas has to be addressed. An approach that is particularly suited for a middleware context since it is based on the behavior of a program as it is observable at its standard input/output interface is proposed in Section 7.2. To implement this approach, group communication is used. An investigation of the responsiveness of a particular group communication system (the Totem protocol) used by this approach is presented in Section 7.3. Based on this generally applicable solution for the input/output problem, the use of replication in Calypso is then described in Section 7.4.

In Chapter 8, the question how to provide a middleware mechanism that can manage resources, namely CPU time, in a way that is compatible with the particular needs of parallel programs is addressed. The mechanism described in this chapter allows resource guarantees for individual programs even in the presence of background load, which is ultimately necessary for assurances about the execution time of a program. Moreover, it also temporally coordinates the distributed execution of programs so that parallel programs do not unduly suffer from this management of resources.

A fourth area that can limit the responsiveness of a distributed program is the communication between its distributed parts. This is an especially important problem in systems that target Wide Area Network (WAN) environments. Such systems promise to remove the limitations on resources inherent in any purely local installation. Charlotte is a member of the Milan family of systems that addresses such an environment. A possibility to specify additional information about a Charlotte program that can be used to considerably increases the efficiency of the program and can also serve as a first stepping stone to predictable execution of parallel programs in even such complex environments is shown in Chapter 9.

Finally, some conclusions from the work presented in this dissertation are drawn in Chapter 10 and perspectives for future research are discussed.

It is very hard to predict, especially the future.
– Nils Bohr

Chapter 2

Service and Responsiveness

In the previous Chapter 1, the intuitive notion of a predictable, timely, and dependable system/service has been used. Here, the possible connotations of this intuition are considered and put in context with existing work on Quality of Service. It is argued that a probabilistic description of service execution is necessary and a general metric, responsiveness, is formalized to reflect this in a uniform way.

2.1 Predictability

In a perfectly predictable as well as dependable system, the future behavior of a system is completely known, the system always delivers the required service, if necessary on time. Additionally, it is possible to give, subject to certain assumptions, a 100% guarantee on this behavior [272].[1] However, the promise of 100% guarantees can actually be considered as a hidden danger or at least misleading: While it is quite clear that such guarantees are only valid as long as the assumptions hold under which they were derived, this truth is not always completely conscious to a developer or user of such a system. Also, no set of assumptions can completely describe the real world in its entirety. It is hence possible that assumptions do not hold in practice, making any 100 % guarantee void. Moreover, since no real system is completely dependable, faults must be included in the system assumptions. It is questionable in how far it is possible to derive absolute guarantees if the existence of faults is considered—e.g., while a system might be completely predictable under the assumption of at most one fault, this assumption itself is artificial: where one component can fail, any component can. Also, attempting to provide such 100 % guarantees can lead to static, inflexible designs derived under a set of restricted assumptions. It has been repeatedly argued that for future systems, such severe restrictions on environment and system assumptions lead to too inflexible a system design (e.g., [188, 272]). All these problems are exacerbated by the increasing complexity of today's systems. Pursuing firm guarantees for a real system can therefore be misleading.

It might thus not be advisable to strive for such absolute guarantees since they might not hold in practice anyway. This is even more so the case since the techniques used to implement such guarantees are often an overspecification of actual needs: It is often not necessary to guarantee all and every requirement of a system; for many aspects, a high probability of meeting the requirements is perfectly sufficient.

Abandoning the illusion of an absolute guarantee and admitting the fact that any statements about a system are only true with a certain probability opens the door to a stochastic description of a system's behavior. Such a probabilistic approach has the chance to balance a system's needs more judiciously and to avoid the traps of overspecification for the sake of pseudo-guarantees; it also has the potential to reduce design time and costs of such systems significantly. Also, since systems are becoming more and more complicated, providing such absolute guarantees is becoming more and more difficult—only statements of a probabilistic nature are possible in the face of growing complexity. Another reason for using a probabilistic metric is that it can also

[1] This concept has been called the "paradigm of guaranteed response" [152]. Section 3.5.3 will discuss in detail an example for a system that follows this paradigm.

handle randomized algorithms or probabilistic assumptions (e.g., assumptions regarding load expressed by a stochastic arrival process). Therefore, a probabilistic metric appears promising. And indeed, the inherent imperfections of real systems have always necessitated a probabilistic description of their dependability, as expressed by probabilistic metrics like reliability or availability (see, e.g., [231]).

The discussion so far has considered the behavior of systems. However, it is by no means clear that the system is the correct level of abstraction to discuss timeliness and dependability. For a user, the system as such is more or less irrelevant; the service the system delivers to a user is the focus of his attention. The behavior of the remainder of the system is irrelevant to a user as long as the used services behave as expected and execute in a correct and timely manner. Moreover, a complex system can easily include a large number of services, each with different characteristics. It therefore appears unreasonable to blur the characteristics of these many services in a description of the entire system, actually losing information in the process. This advocates a service-centered viewpoint for a metric of timeliness and dependability as opposed to a system-centered one (an extended discussion of service versus system as abstraction level and probabilistic metrics can be found in [302]). Such a metric should be able to argue about quantitative properties of a service and possibly allow a comparison between them.

Summing up, a metric is needed that expresses an essential property of a service: its correct and timely completion. For the reasons given above, such a metric can only be a probabilistic one. Therefore, a minimal requirement for this metric is the capability to argue about the probability with which a service is completed correctly at what time. This probability is an essential characteristic of the user-perceived quality of a service.

In the following Section 2.2, it is considered how the much-used term Quality of Service might fit to the present context of timely and dependable execution of parallel programs. It will also consider some specific examples of attributes of service quality and find them unsuitable. As a proper Quality-of-Service attribute, the notion of responsiveness, introduced in [188], is described in Section 2.3 and refined to adapt it to the present need of a metric capable of expressing probabilistic properties of a service. But first, some brief remarks on terminology seem to be in order.

In this dissertation, the following terminology shall be adopted (following [221, 266]). Quantifiable characteristics or properties are called *metrics*, e.g., weight and length are metrics of a physical object. Informally speaking, a metric is what one would really like to know. Typically, a metric is only well defined if used in a defined context, e.g., the length typically depends on the temperature and is therefore only well defined for any given temperature.

Measure is a reference standard or sample used for the quantitative comparison of properties, e.g., the meter and the inch are measures of length. *Unit* is a synonym for measure. Only measures in accordance with the International System of Units (SI) [281] will be used, with decimal modification if convenient (such as kilo, mega, giga, centi, milli, micro), also as predicated by the SI. For measures like byte, which are used with prefixes of the power of 2, the SI proposes prefixes kibi, mebi, gibi corresponding to 2^{10}, 2^{20}, 2^{30}, respectively. The intent is to avoid confusion with modifiers referring to powers of 10 [29]. However, since this is not yet a commonly accepted practice, the usual prefixes will be used, and additional information will be given if ambiguities might arise.

A *measurement* is both the act of measuring a given metric as well as the actual numerical value determined by measuring. Hence, 10 mm = 0.01 m is a valid measurement of the length metric of a physical body.

2.2 Quality of Service

One of the driving forces behind the work on Quality of Service (QoS) are new kinds of applications, in particular, multimedia applications that have real-time requirements—examples include today digital voice and video, in the future perhaps television, medical diagnosis, or multimedia conferencing. In such a real-time application, some services must be completed by or at a given time. In the present context, the service “execute parallel program” in an environment of possibly faulty, COTS-based workstations is under consideration; its quality metric is yet to be defined. What are the standard metric definitions of Quality of Service?

Due to the many facets and uses of the term Quality of Service, there is no uniform definition or usage of it. The term originated in the networking community and originally described properties of data communication services, mostly for non-time-dependent traffic. Today the idea of a guarantee for certain metrics is intimately linked with the concept of Quality of Service.

Some standardization bodies have proposed definitions for Quality of Service:

- The International Telecommunication Union (ITU) defines Quality of Service as "a set of quality requirements on the collective behavior of one or more objects" [116] (a definition that is also adopted by [114]), in [115] as "the collective effect of service performance which determine the degree of satisfaction of a *user* of the *service*" and continues to note "The term 'quality of service' is not used to express a degree of excellence in a comparative sense nor is it used in a quantitative sense for technical evaluations. In these cases a qualifying adjective (modifier) should be used."

- The Internet Engineering Task Force is concerned (in the context of Quality of Service) with real-time applications like audio- or video-playback over the Internet. In [41], the most important aspect of Quality of Service is control over end-to-end packet delays.

- In the context of the Reference Model for Open Distributed Processing (RM-ODP) and TINA-C, LEYDEKKERS et al. [174] address the description of Quality-of-Service specifications for *continuous* data flows in the computational and engineering viewpoints. This paper focuses on " ... the QoS contract in relation with real-time interaction for continuous flows such as audio and video. A QoS contract provides a specification of the service provided and 'level of service' agreed between the involved computational objects."

Owing to their broad range, none of these standard definitions quite matches the intuition for a service that predictably and dependably executes programs. A specialized definition is necessary, in the sense of the "qualifying adjectives" of [115]. Such qualifiers abound, and a proper one must be chosen.

Recently, the term "end-to-end service availability" has created some amount of interest. The intuitive notion connected with it is the need to consider all levels in a system/service hierarchy to be able to provide highly dependable services to the user and that it is not sufficient to concentrate only on some isolated system levels (e.g., hardware).

This notion of end-to-end service availability is commonly used in the telecommunications industry, and an example from this field shall serve as a basis for discussion. The typical service in telecommunications is to enable customers to make phone calls. The two ends between which the service is delivered are quite clear: the caller and the callee. Following the definition of SIEWIOREK and SWARZ [260]: "[Availability is] a function of time, $A(t)$, [which] is the probability that the system is operational at the instant of time, t ... Availability is typically used as a figure of merit in systems in which service can be delayed or denied for short periods without serious consequences", service availability is then the instantaneous probability that at any given time t, the call between caller and callee works.

Such a call service is actually implemented on top of many little services: transmit a single packet of digitized speech from one end to the other. The overall length of such a lower-level service is small, even insignificant compared to the total length of a call. Additionally, it is more or less irrelevant if such a single packet is lost. Therefore, the call service can be "denied for short periods without serious consequences." This allows to consider instantaneous availability as a Quality-of-Service metric for phone calls. Moreover, a phone call has no inherent termination time. It is open ended and behaves more like a stream of data.

This discussion can also be cast in terms of call establishment: Service availability is then the probability that a call will be put through after the number has been dialed. Call establishment can be considered as an instantaneous event and therefore no attention needs to be given (in a first approximation) to the time it takes to actually establish the call.

However, such a scenario does not quite match a (parallel) execution of a program. While a parallel program may consist of routines the execution of which might be denied (if the programming environment is capable of handling such situations), eventually all parts must be executed. What is more, quite unlike a

phone call, the termination time of a (parallel) program is usually important and should be as short as possible or before a given, pre-specified time.

Therefore, rather than a metric that only considers a single point in time, a metric is needed that takes into account both the starting time of a service request and its finishing time. This ability to consider these two times independently is an important requirement for a metric for execution of programs. Such a metric can be found in the notion of responsiveness, which will be discussed in the following section in detail.

2.3 Responsiveness

In the previous Section 2.2, the need for a metric that formalizes the intuitive concept of timeliness and dependability has been identified. Such a metric must take into account both the probabilistic nature of services in a complex system as well as the nonzero execution time of any service request and, hence, the need to talk about service request time and service completion time. This section defines such a metric: responsiveness, introduced by MALEK [188, 189] and later extended by WERNER and KARL [303].

2.3.1 Definition of responsiveness

The primary concern is a metric that provides information about the dependence of service request and service completion for a given service executing on a given system. The following Definition 2.1 introduces the term response set RS for this purpose.

Definition 2.1 (Response Set). *Given a service* Srv *on a system* Sys*, the* response set RS *is defined as*

$$\mathrm{RS}_{\mathrm{Srv},\mathrm{Sys}} \stackrel{\mathrm{def}}{=} \begin{cases} \mathbb{R} \times \mathbb{R} \to [0,1] \\ (t_{\mathrm{req}}, t_{\mathrm{resp}}) \mapsto r \end{cases}$$

such that an invocation of Srv *at* t_{req} *will be completed correctly at or before* t_{resp} *with probability* r.[2] *If both* Srv *and* Sys *are clear from context, they can be omitted.*

Evidently, this definition hides a certain amount of implicit information in Srv, Sys, and t_{req}:

- A service invocation defines the service itself and the parameters of this particular instance, which is important for the resource requirements (like runtime, memory and so on).
- The system characterizes the available resources (like CPU speed or number of nodes), but also the fault model assumed for this particular system. Also, the system itself can be dynamic: resources (e.g., nodes) may become available or may be withdrawn. These are important sources of probabilistic behavior.
- The time t_{req} at which a service is invoked also specifies implicitly the state of the system at this time (e.g., the current load of the system).

Any actual evaluation of RS has to make these assumptions explicit and take them into account.

In some contexts, inverse functions of RS can be of interest. For example, it might be possible to ask: "given a service, a start- and end-time, what is the required system configuration (i.e., how many nodes are needed) to achieve a certain probability of completion within this time?" or variations thereof. This information is clearly represented by the response set as its inverse functions. The formulation of RS as function was only chosen for its simplicity; an equivalent formulation of RS as a 3-tuple could be used to make the mutual dependency of the five parameters more explicit.[3]

[2] Obviously $\mathrm{RS}(t_{\mathrm{req}}, t_{\mathrm{resp}}) = 0$ for $t_{\mathrm{resp}} < t_{\mathrm{req}}$.

[3] Define RS as $\mathrm{RS}_{\mathrm{Srv},\mathrm{Sys}} \stackrel{\mathrm{def}}{=} \{(t_{\mathrm{req}}, t_{\mathrm{resp}}, r) |$ an invocation of Srv on system Sys at t_{req} will be executed correctly by t_{resp} with probability $r\}$.

In many environments, it is not sufficient only to execute a service; rather, the completion of the service by a given time should be guaranteed. Results are worthless if they are not produced by a given time (commonly called a deadline, see Section 3.4 for details). As has been discussed before, firm guarantees are unrealistic; rather, the probability of meeting this deadline should be the measure of choice.[4]

Considering such a service with a specific deadline, the response set can be specialized to the responsiveness function:

Definition 2.2 (Responsiveness Function). *Let* Srv *be a service with a relative deadline* d*, executed on system* Sys*. The* responsiveness function RF *is defined as*

$$\mathrm{RF}_{\mathrm{Srv},\mathrm{Sys}}(t_{\mathrm{req}}) \stackrel{\mathrm{def}}{=} \mathrm{RS}(t_{\mathrm{req}}, t_{\mathrm{req}} + d)$$

RF simply points out the importance of the deadline, ignoring the possibility of earlier termination, but highlights the fact that the probability of correct completion by a given deadline may well vary during the life of a system. In particular, $\mathrm{RF}_{\mathrm{Srv},\mathrm{Sys}}(t_{\mathrm{req}})$, the responsiveness function for a given service request Srv happening at some time t_{req}, can be called the responsiveness of this particular service request.

Some applications have timeliness requirements more complicated than just a single deadline. Consider as an example a service that must be completed at the latest t_{late} time units after it has been requested (a relative deadline), but that it is also not permitted to complete earlier than t_{early} time units after request—the service must hence complete in the time window $[t_{\mathrm{req}} + t_{\mathrm{early}}, t_{\mathrm{req}} + t_{\mathrm{late}}]$. The probability of a service meeting such a timing requirement is $\mathrm{RS}(t_{\mathrm{req}}, t_{\mathrm{req}} + t_{\mathrm{late}}) - \mathrm{RS}(t_{\mathrm{req}}, t_{\mathrm{req}} + t_{\mathrm{early}})$.

Another possible simplification arises if the dependency on the time of request is dropped. To justify this simplification, the service itself and its resource requirements must not depend on the actual time of request, and the system must behave uniformly over the entire time. In practice, this is usually not the case—consider, e.g., the increasing fault rates of aging hardware. But if, e.g., only a very simple load model and a constant fault rate are assumed, then this might be an acceptable simplification. Formally:

Definition 2.3 (Response Time Distribution). *If a service* Srv *executed on system* Sys *satisfies*

$$\forall t_1, t_2, \Delta \geq 0 : \mathrm{RS}_{\mathrm{Srv},\mathrm{Sys}}(t_1, t_1 + \Delta) = \mathrm{RS}_{\mathrm{Srv},\mathrm{Sys}}(t_2, t_2 + \Delta),$$

then this service Srv *is called* stationary *with respect to* Sys *and the* response time distribution RTD *of* Srv *on* Sys *is*

$$\mathrm{RTD}_{\mathrm{Srv},\mathrm{Sys}}(t) \stackrel{\mathrm{def}}{=} \mathrm{RS}_{\mathrm{Srv},\mathrm{Sys}}(0, t)$$

Definition 2.3 has been suggested in [303] as a possible foundation for responsiveness. It is important to reiterate that using the RTD is only valid under simplified assumptions and does not apply in general. Also note that in general a response time distribution is not a probability distribution since there is no reason to assume that $\lim_{t\to\infty} \mathrm{RTD}(t) = 1$. Nonetheless, the response time distribution (in the following also called the runtime distribution if a service corresponds to the execution of an entire program) is a convenient metric for many discussions, as the start time of a request can indeed be neglected in many circumstances.

If one accepts the difficulties with this simplification and assumes a stationary service, and additionally considers a service associated with a deadline, then one can define the responsiveness of a service on a given system (for non-stationary services, only the responsiveness of a particular service invocation $\mathrm{RF}_{\mathrm{Srv},\mathrm{Sys}}(t_{\mathrm{req}})$ can reasonably be defined):

Definition 2.4 (Responsiveness of a service). *Given a stationary service* Srv *with deadline* d *on system* Sys*, the* responsiveness *of this service is*

$$r_{\mathrm{Srv},\mathrm{Sys}} \stackrel{\mathrm{def}}{=} \mathrm{RTD}_{\mathrm{Srv},\mathrm{Sys}}(d)$$

[4] This is indeed a generalization on the paradigm of guaranteed response: as long as for a given system it can be shown that all deadlines are met with probability 1, the system does conform to this paradigm.

For a stationary service, $r_{\mathrm{Srv,Sys}} = \mathrm{RTD}_{\mathrm{Srv,Sys}}(d) = \mathrm{RF}_{\mathrm{Srv,Sys}}(0)$.

To sum up, in general the response set RS is the metric of choice to assess end-to-end usefulness of a service to a user; for real-time services the responsiveness function serves as a reasonable simplification; and under proper assumptions only the response time distribution needs to be considered. For brevity's sake, "responsiveness" shall henceforth be used to collectively refer to the definitions introduced in this section. A similar abbreviation is the usage of "responsive system": strictly speaking, any system is responsive (after some time, all services finishes successfully with a—perhaps low—probability), but in its lax use, the term usually refers to systems with rather highly responsive service execution.

2.3.2 Related approaches

At a first glance, this definition of responsiveness does not seem to add anything new to other, existing notions. In particular, imprecise computation [183] and performability [201, 202] come to mind.

Imprecise computation

In imprecise computation [183], a service execution increases in value the longer it lasts. For example, a numerical approximation becomes more precise the more iterations are computed. Hence, even if a service is not completed, it still has incurred some value. This is usually expressed by utility functions, which map the execution time of a service to its value (e.g., between 0 and 1, but not necessarily so). Imprecise computation is a valuable approach in settings were timely, approximate results are better than late, precise results; image processing or automated target tracking are mentioned as examples for such settings in [183]. But responsiveness behaves differently: here a service has to be completed, otherwise it does not incur any value at all. However, the actual length of execution is only known probabilistically.

Both imprecise computation and responsiveness can be unified into a concept more general than either of the two.

Definition 2.5 (Imprecise Response Set). *Given a service invocation* Srv *on a system* Sys*, the* imprecise response set IRS *is defined as*

$$\mathrm{IRS}_{\mathrm{Srv,Sys}} \stackrel{\mathrm{def}}{=} \begin{cases} \mathbb{R} \times \mathbb{R} \times [0,1] \to [0,1] \\ (t_{\mathrm{req}}, t_{\mathrm{resp}}, v) \mapsto r \end{cases}$$

such that an invocation of Srv *at time* t_{req} *has incurred at least value* v *by* t_{resp} *with probability* r *(monotonicity of the imprecise value function is assumed).*

The imprecise response set is more general than the response set from Definition 2.1 since it takes relative values for incomplete executions into account. It is also more general than imprecise computation since it can consider probabilistic behavior of service execution.

It would indeed be possible to model this IRS with pure responsiveness alone, but for a price. The basic observation is the fact that a service (reasonably) only increases in value in discreet steps—when a loop iteration has completed, for example. Such an elemental step could be considered as a micro service, and the number of completions of this micro service can be counted; the response time distribution of the service is then the convolution of the RTD of the micro service. However, this complicates analysis and potentially introduces a multitude of pseudo services. Also, it only works if the service can be easily dissected into simpler services.

To illustrate the IRS concept, consider a service that has five possible values, e.g., precision levels, according to the imprecise computation model. The time it takes to upgrade the service's value is geometrically distributed with parameter 0.99.[5] Figure 2.1 shows the imprecise response set for this service; curves indicate the probability to have reached at least value $1, \ldots, 5$, respectively.

[5] A geometric random variable represents the number of draws of a standard uniform random variable that are needed before the sum of these draws is larger than the parameter of the geometric random variable.

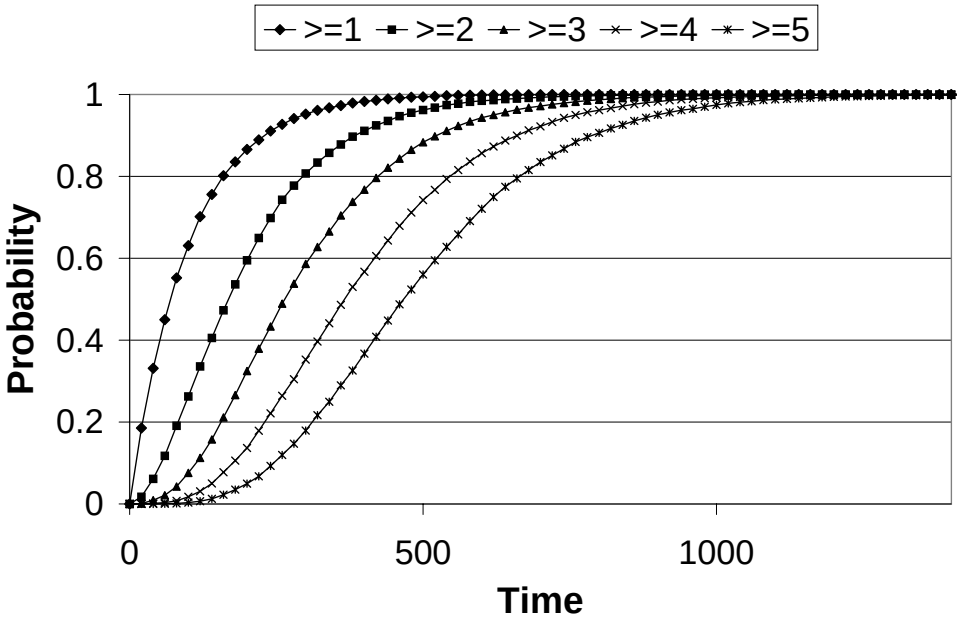

Figure 2.1: Imprecise response set for a service with five different values (number of steps necessary for reaching the next value level is geometrically distributed with parameter 0.99) shown as probability over t_{resp}, $t_{\mathrm{req}} = 0$.

Performability

Performability [201, 202] addresses combining performance and dependability issues in modeling degradable computer systems. A computer system is degradable when the system can exhibit a number of different user-discernible levels of performance during the course of a utilization period; a simple example is throughput in a multi-processor environment. In such a system, neither performance nor dependability can be considered in isolation from each other. Performance evaluations made for the fault-free system usually do not reflect the performance of the system if some parts of it have failed. On the other hand, reliability estimations have to take into account the system's capability of continued, yet degraded operation even after parts have failed. Performability integrates these two aspects.

For a performability description, levels of accomplishment are identified with sets of possible traces of system executions over time. These traces represent the different possible state sequences the system can undergo (e.g., because of faults) and allow a very flexible modeling of different perspectives of a system by employing different metrics in a general performability framework.

Comparing performability with responsiveness shows that they represent two different approaches to a similar problem. Both are concerned with the user's perspective on a system or service, respectively. One obvious difference is that responsiveness explicitly focuses on the behavior of a service in real time. It is possible to model real time in performability also, but such a modeling requires additional definitions about the relationship of accomplishment levels and real time. Also, responsiveness as such has no explicit notion of a degraded system or service: a service has to complete entirely before any value is assigned to it—the discussion of imprecise response sets has shown how this could be generalized. Potential failures of a degradable system and their impact on performance are implicitly, yet precisely represented by the response set since these failures are part of the probabilistic description of the system and hence modify the probabilities of successful service completion after a given time. This representation requires, however, that the system is designed so that the only user-discernible consequences of failures are delayed service execution or complete service failure. From this perspective, responsiveness can be regarded as one particular performability metric.

A more profound difference is the level of abstraction: responsiveness deals with individual services, performability concentrates on systems, commonly made up of at least a few services that in cooperation achieve

the degradability. Hence a better comparison would be between performability and an extended definition of responsiveness that considers a service graph: different possibilities to complete a service, each with an associated level of accomplishment. Such an extended notion of responsiveness would in turn generalize performability owing to the explicit consideration of real time.

2.3.3 Some examples

The abstract definitions of Section 2.3.1 are here illustrated with a few, brief examples. The examples in this section are somewhat constructed; Chapters 5 and 6 contain complete problems studied with respect to responsiveness.

Service execution on an unreliable system

Suppose a service with a fixed execution time t_{S} is to be executed on an unreliable system. Let X denote the lifetime of this system and $F_X(t)$ the cumulative distribution function of this lifetime [283, p. 118]. A service execution requested at time t_{req} will complete at time $t_{\mathrm{req}} + t_{\mathrm{S}}$ if the system survives until that time (if no repair is assumed and the system is operational at time 0). Hence, the response set of this service is

$$\mathrm{RS}(t_{\mathrm{req}}, t_{\mathrm{resp}}) = \begin{cases} 0 & t_{\mathrm{resp}} < t_{\mathrm{req}} + t_{\mathrm{S}} \\ \Pr(X > t_{\mathrm{req}} + t_{\mathrm{S}}) & \text{else.} \end{cases}$$

The examples becomes more interesting if the execution time of the service is only known probabilistically, e.g., as the density function $f_S(t)$. Any particular service execution time τ appears with probability $f_S(\tau)$, and the probability to complete the service (requested at time t_{req}) in this case is then $\Pr(X > t_{\mathrm{req}} + \tau)$, analogously to the case above. By the law of total probability, it is possible to sum these probabilities, weighted by their probability of occurring, to obtain the total probability to complete at service, requested at time t_{req}, at or before time t_{resp}:[6]

$$\mathrm{RS}(t_{\mathrm{req}}, t_{\mathrm{resp}}) = \begin{cases} 0 & t_{\mathrm{resp}} < t_{\mathrm{req}} \\ \int_{\tau=0}^{t_{\mathrm{resp}} - t_{\mathrm{req}}} f_S(\tau) \Pr(X > t_{\mathrm{req}} + \tau) \mathrm{d}\tau & \text{else.} \end{cases}$$

A similar approach will be used in a much more extensive analysis of a scheduling mechanism in Chapter 5.

Internet-related services

Anybody who has used the Internet repeatedly has undoubtedly noticed a strong dependence of response times on the time of day and day of the week. While in the early morning, connections are typically good, they deteriorate notably during the day. This time dependence is a well known effect for traffic in wide area networks. Interestingly enough, Internet traffic has a chaotic, self-similar characteristic for which Poisson modeling is unsuitable [222]. Such services are obvious and relevant examples of non-stationary behavior.

Design example

In joint work with M. Werner [303], the ideas of responsiveness are discussed using a design example: given a task graph with dependencies between the tasks, probabilistically distributed task execution times, and two potentially failing processors, what is a good scheduling algorithm to execute this graph before its deadline has expired? Using the response time distribution as metric, a naïve scheduler, a simple hot standby, a distributed approach and eager scheduling are compared. The respective response time distributions for these schedules clearly show the necessity to consider both fault-tolerance and real-time properties in an integrated fashion.

[6] It is assumed here that negative execution times are not possible and that the service execution time does not depend on t_{req}.

2.3.4 Challenges of responsiveness

Responsiveness or, more precisely, the response set and response time distribution, are very attractive metrics owing to their concise representation of a service's probabilistic behavior in real time and in the presence of faults. However, it does have its own particular set of challenges as well.

The complexity challenge

Computing a response time distribution is usually a very complex undertaking. The complexity or even possibility of solving this problem heavily depends on the chosen modeling technique and on the desired precision of the results. A plethora of modeling techniques is available; examples include, but are not limited to, Markov models and queueing theory (in particular, real-time queueing [170]), Petri nets, methods based on temporal logic, or combinatorial approaches. A common problem is that for all of these methods, deriving random distributions for any non-trivial problem is difficult; often, only means and perhaps some stochastic moments are derivable. Sometimes it is possible to find expressions for random distributions using such a technique; however, this expression might only be solvable numerically. Sometimes, only simulation results are obtainable, and sometimes, it is possible to characterize a service execution by using actual measurements.

The definitions given above are orthogonal to the time of their actual calculation or estimation. This calculation can—in theory—happen on-line or off-line. In an off-line calculation the state of the system (as far as load and/or faults are concerned) is only known probabilistically. In an on-line calculation, such as the calculation of the responsiveness function for a service request, the state of the system at this point in time is known (to a certain degree) and this knowledge can be used in the responsiveness estimation. Therefore, such an on-line calculation is potentially more precise than an off-line one. From a theoretical point of view, this corresponds to the computation of conditional probabilities.

An on-line calculation also has the possibility to monitor the actual progress of a service execution and use this information, e.g., to make decisions about resource allocation, fault-tolerance protocols, admission tests for additional load, or to take into account the impact of various system modes. For an accurate on-line monitoring, the service might need to cooperate and provide status information in a specified manner. Also, rapid estimation techniques for RS are necessary.

The comparison challenge

Often, people are interested in a comparison of different alternatives to implement a service or in the relative quality of two systems. More precisely, the question is how to compare the response sets or, somewhat simpler, two response time distributions of a given service on two different systems.[7]

Non-stationary services with a meaningful deadline[8] can be compared using their request times t_0 and $\mathrm{RF}(t_0)$. If the service is additionally stationary (on both systems), the responsiveness $\mathrm{RF}(0)$ is the natural basis for comparison: it is just a number. If no deadline is given for a stationary service, one has to compare the response time distributions as functions. Formally, how can an order relation be defined on the set of response time distributions so as to reflect an intuitive notion of "better (or more responsive) service"? While the function space has a partial order relation[9] it is by no means clear how to judge two functions that are non-comparable with respect to this partial order.

A simple metric to compare functions with this generality would be the mean of the RTD. But since responsive service execution is the objective, the mean alone is insufficient. An obvious supplement is the variance or standard deviation. Yet even with these two numbers the ugly fact of incomparability rears its

[7] Note that this is not precisely identical to comparing two systems, since it is not necessarily the case that two different systems allow the same set of services. On an end-to-end level, however, this makes sense nonetheless, since available services can be regarded as a user requirement; similarly, both systems must allow the same service request times.

[8] A meaningful deadline is one that reflects some actual requirement imposed by the system's environment. Any deadline used only internally in a system is purely artificial and should not be considered as a basis for comparison of a system's or a service's level of merit.

[9] For two functions f_1, f_2 that both map a domain D to, say, the real numbers $\mathbb{R}$, $f_1 < f_2$ if and only if $\forall x \in D : f_1(x) < f_2(x)$. It is only a partial order, since there are functions f_1, f_2 such that $\exists x_1, x_2 \in D : f_1(x_1) < f_2(x_1) \wedge f_1(x_2) > f_2(x_2)$.

head: is a small mean with large variance or a larger mean with a smaller variance preferable? In such a situation, the more predictable service might be preferable.

What characterizes a predictable service in the first place? Ideally, it would be a service with a precisely determined runtime t_0, i.e., with a response time distribution of the form $\mathrm{RTD}(t) = H(t - t_0)$.[10] Such a service is obviously unrealistic, but a tractable definition for "more predictable" can be generalized from it—intuitively, the fewer possibilities for runtimes there are, the better:

Definition 2.6. *Let r, r_1, r_2 be response time distributions. Define $\mathcal{D}(r) \stackrel{\text{def}}{=} \{x | \forall \epsilon > 0 : \exists x_1 \in (x-\epsilon, x+\epsilon) : r(x) \neq r(x_1)\}$ as the differential set of r, the set of points where r changes its value.*

The response time distribution r_1 is called more predictable than r_2 if and only if:

- *$\mathcal{D}(r_2)$ is uncountably infinite, $\mathcal{D}(r_1)$ is uncountably infinite, and $\mathcal{D}(r_1) \subset \mathcal{D}(r_2)$, or*
- *$\mathcal{D}(r_2)$ is uncountably infinite, $\mathcal{D}(r_1)$ is at most countably infinite, or*
- *$\mathcal{D}(r_2)$ is countably infinite, $\mathcal{D}(r_1)$ is countably infinite, and $\mathcal{D}(r_1) \subset \mathcal{D}(r_2)$, or*
- *$\mathcal{D}(r_2)$ is countably infinite, $\mathcal{D}(r_1)$ is at most finite, or*
- *$\mathcal{D}(r_2)$ is finite, $\mathcal{D}(r_1)$ is finite, and $|\mathcal{D}(r_1)| < |\mathcal{D}(r_2)|$, or*
- *$\mathcal{D}(r_2)$ is finite, $\mathcal{D}(r_1)$ is finite, and $|\mathcal{D}(r_1)| = |\mathcal{D}(r_2)|$ and $\bar{\mathcal{D}}(r_1) < \bar{\mathcal{D}}(r_2)$, where $\bar{\mathcal{D}}(r)$ is the variance of the set of changes of r with*

$$\bar{\mathcal{D}}(r) \stackrel{\text{def}}{=} \mathrm{Var}\{\lim_{x \to x_0, x > x_0} r(x) - \lim_{x \to x_0, x < x_0} r(x) | x_0 \in \mathcal{D}(r)\}$$

(intuitively, a service with a large jump and a few small ones is more predictable than one with many jumps of the same size).[11]

Unfortunately, $\mathcal{D}$ is very difficult to assess for a real service executed on a real system. Therefore, often the standard deviation, and perhaps the variation coefficient (the standard deviation divided by the mean), are the only practical metrics of predictability. Yet these characterizations have to be used with care: it is possible to construct two response time distributions r_1, r_2 such that r_1 is more predictable than r_2 in the sense of Definition 2.6, but the variation coefficient of r_1 is larger than that of r_2. Nevertheless, the variation coefficient is a suitable approximation for a predictability metric for practical purposes. A precise metric for comparing two arbitrary services is still a question of future work.

Note also that predictability in the sense of Definition 2.6 or in the sense of a small standard deviation is a different concept than responsiveness, since no concept of a deadline is considered in the definition of predictability. A service might be highly predictable, but also miss a deadline; it might meet all deadlines, but have a large standard deviation (or variation coefficient) or a large $\mathcal{D}$.

The composability challenge

An important desideratum for abstractions of computer systems is composability. Today's system are hardly ever completely designed from scratch, but are rather composed of pre-existing components. Given some properties of these components and the way they are composed, what are the properties of the new, composed system?

This question touches on some critical points. One is to identify a relevant set of properties, another is to characterize the process of putting systems together so that it is both effectively and efficiently possible to argue about some properties of the composed system, a third is to see if the new system has some properties

[10] H is the Heaviside function $H(t) = 1$ for $t \geq 0$ and $H(t) = 0$ for $t < 0$.

[11] For this last case, note that the finiteness of $\mathcal{D}(r_1)$ implies that all points were r_1 changes its values are points of discontinuity of r_1 (and similar for r_2).

that were not present in its individual parts (an example for such an emerging property would be fault tolerance in a system based on potentially faulty components).

For responsiveness as an example for such a property, the question of composability can be made more concrete. A service is commonly composed of a number of lower-level services. Given the responsiveness of these services, what is the responsiveness of the composed service? If an efficient technique can be found to compute the responsiveness of composed services under reasonable assumptions about the interactions of the lower-level services, responsiveness can be considered as an example of a composable abstraction. The search for such a technique is the objective of current research in the Computer Architecture and Communications group at Humboldt-University Berlin and not the topic of this dissertation.

The challenge of Commercial Off-The-Shelf systems

Closely related to the question of composability is the relationship of responsiveness and COTS systems. By definition, COTS systems consist of existing components, but they aggravate the problems of responsiveness since often only insufficient knowledge about both system and components is available (e.g., scheduling policies of a COTS operating system are not specified, making any analysis difficult). Moreover, COTS systems are often open systems, making fixed assumptions about the load or the way individual services interact with each other impossible.

This very openness requires some mechanisms to tame the interactions of services in a manner consistent with the COTS nature of the system. This disallows, e.g., any changes to the existing system (be it hardware or operating system) and requires middleware solutions put on top of these systems. One example for such a middleware is controlled arbitration of resources, for which a solution is described in Chapter 8; Chapter 7 introduces a concept for a replication-based fault-tolerance mechanism that conforms to this necessity for interface-respecting middleware solutions. However, responsive COTS systems are a problem area that still requires a lot of research and, owing to its enormous complexity and number of interacting factors, does not promise any simple, final solutions in the foreseeable future.

"The time has come", the walrus said,
"To talk of many things:
Of shoes—and ships—and sealing wax—
Of cabbages—and kings—
And why the sea is boiling hot—
And whether pigs have wings."

– Lewis Carroll

Chapter 3

Related Work

Distributed systems in general and cluster-based systems in particular have been an object of research for a considerable time. Some notable projects are described in this chapter, mainly considering projects that focus on performance, fault tolerance, real time, or Quality-of-Service aspects. The more interesting ones of these projects have more than a single focus; in particular, some examples for systems combining fault-tolerance and real-time capabilities in the sense of responsiveness are discussed.

3.1 Kinds of clusters

Cluster-based systems have been built for a number of different purposes. Perhaps the historically first are systems that are mostly concerned with executing computation-bound, large-grained parallel applications. Such applications rarely communicate between concurrent parts and are therefore ideally suited for execution even on loosely coupled clusters. Examples for such systems include Condor [181] or Utopia [312]. They excel at achieving throughput for the entire system, but often do not provide exceptional performance or guarantees on performance for an individual program

Dedicated high-performance clusters are build to execute a parallel program as fast as possible. The main challenge here is communication performance; but the lack of a single system image (compared to supercomputers) adds other problems as well. Examples for these kind of projects truly abound—an overview can be found in, e.g., [111, 237]. Some notable examples for these two kinds of cluster organization (high throughput and high performance) are discussed in Section 3.2 along with some systems that extend beyond clusters to wide area environments.

Another kind of cluster-based systems is concerned with providing high availability, but do usually not target parallel applications. Here, a user should be able to rely upon the permanent availability of the system to execute user requests. Traditionally, this availability can be achieved by hot or cold standby techniques such as primary-backup, often implemented with custom networks and operating systems. Some of the commercial systems from Tandem, Stratus or Sequoia (cp., e.g., [260]) belong to this category, although one would hesitate to classify these systems as COTS clusters since they do heavily modify the basic systems or a truly custom-designed. More in the spirit of COTS are systems like Wolfpack [257] that extends Windows NT yet runs on typical COTS clusters; details can be found in Section 3.3.

Research in real-time systems has, to an even larger degree than fault-tolerance research, focused on custom-built systems. This focus is dictated by the complexities of commodity systems with regard to their behavior in real time. A few projects are discussed in Section 3.4, including one that addresses real-time

capabilities for high-performance computing. Addressing both real time and fault tolerance is the objective of projects described in Section 3.5.

The notion of Quality of Service is intimately linked with the idea of a guarantee (in whatever form) on the service quality, and as such includes real time as a special case. Much research has been performed in the context of Quality of Service; a brief overview of this area is given in Section 3.6, considering network Quality of Service and endsystem Quality of Service in turn.

All these kinds of cluster organizations (or custom-designed architectures) can be regarded as first-generation approaches. For a second generation, the combination of high availability with high performance and guaranteed Quality of Service exhibits the union of all the problems mentioned before: A collection of (possibly heterogeneous) machines should be usable under an abstraction of an ideal, reliable, predictable machine. Such a system provides reliable and guaranteed execution of (multiple) programs. No current system has all these capabilities, but it is the ultimate goal of many research efforts.

3.2 Focus on performance

Achieving high throughput or high performance for parallel programs executing on a cluster of workstations is a problem with many facets. It includes more general questions regarding the communication system between nodes and the architecture of an individual node as well as more specific problems regarding the actual parallel programming of such clusters. Since many supercomputer designs are today based on standard microprocessors, the architecture of the individual node is not that much different and performance differences between clusters and supercomputers mostly stem from the communication network.

Possible candidates for communication networks for a cluster are discussed in the following Section 3.2.1. Within a node, an important point is the integration of the network interface into the host architecture and how application software can access the network (see Section 3.2.2 for details). While this is to some degree relevant for any kind of cluster environment, high-performance computing is additionally faced with the choice and implementation of suitable programming models—see Section 3.2.3. Finally, in Section 3.2.4, some projects that attempt to reach beyond local clusters and harness the Internet for wide area parallel computing are discussed. A more exhaustive list of projects and systems focusing on parallel computing can be found in [111, 237]

3.2.1 Communication networks

The prevalent LAN technology today is classic 10 Megabits per second (Mbps) Ethernet and its newer version, Fast Ethernet with 100 Mbps bandwidth; token ring-based systems are also to be found. The main advantage of these technologies is their low price and their general availability at most sites. Consequently, low-cost clusters like Beowulf [240] are based on this technology. However, Ethernet or similar technologies is not sufficient for truly high-performance computing—supercomputer networks are about one order of magnitude faster. LANs typically do not provide reliable delivery of messages, which must be compensated for by time-intensive higher protocol layer processing. Ethernet suffers additionally from saturation of the network (unless expensive switches are used) and is unable to give Quality-of-Service guarantees. This has to be addressed with additional protocols such as Resource Reservation Protocol (RSVP) [42] or the currently introduced Differentiated Services framework [36], for which expensive intermediate routers are needed. In a nutshell, these technologies are quite attractive and often adequate for high-throughput computing due to their low price and simplicity, but are, unfortunately, insufficient for many high-performance applications.

Gigabit Ethernet [87] is a follow-up technology on traditional Ethernet and is currently introduced in commercial products. It promises to provide 1 Gbps bandwidth combined with easy migration from standard Ethernet, low costs and the capability to provide support for new types of applications like multimedia. However, since it is basically a faster type of Ethernet, it does not offer any support for Quality of Service in the data link layer.

Asynchronous Transfer Mode (ATM) [4, 165, 290] seems more promising, even more so since it provides the capability to guarantee certain network service parameters (like bandwidth and latency). With its

telecommunications background, ATM networks are easily capable of spanning large physical distances and scale well to large amounts of nodes. But it has been convincingly argued [63] that the network interfaces available for ATM networks have too high overhead to be useful for parallel processing; WELSH et al. [301] report user-level round-trip times of 60 μs over FastEthernet and 90 μs with ATM. Also, ATM interface cards and switches are still rather expensive.

The communication subsystems used in supercomputers are all switch-based with very low error rates on the order of 10^{-15} errors/packet. They also offer flow control mechanisms (which are realized on a link-by-link basis in hardware and not in higher protocol layers on an end-to-end basis like in TCP). To employ these network technologies in a LAN environment, low error rates have to be provided over a somewhat larger physical distance (about 25 m). So-called system area networks [109] meet these requirements. Nectar [277] is an early research example, Myrinet [39] was one of the first commercially available products. It provides very low error rates in combination with latencies of about 1 μs between application processes and bandwidth of about 125 Mbps in its original version, the current version offers 1.28 Gbps. Newer versions are expected to achieve up to 10 Gbps in the near future [51]. Myrinet also uses wormhole routing, which is normally only found in supercomputer networks. On top of Myrinet, some large configurations for parallel clusters have been built, notably a cluster with 192 processors based on Windows NT workstations [52].

Other SAN examples include Scalable Coherent Interface (SCI) [112] or Compaq's ServerNet [17, 252], the latter is mostly intended for interconnecting high-performance servers and is also about one order of magnitude more expensive than, e.g., Myrinet. ServerNet is planned to be transparently integrated in the standard networking Application Programming Interface (API) of the Microsoft Windows NT operating system [58] and a growing impact of ServerNet can therefore be expected.

Both the available bandwidth and the latency of SANs is quite competitive with supercomputer networks (see Table 1.1). Low latency is often more relevant to high-performance applications than high bandwidth, since typically small packages have to be sent (especially for fine-grained parallel programs). These networks have to be employed judiciously, though. For example, the low error rates provided by them are ill matched with protocols such as TCP/IP used in most COTS systems. TCP has been designed as a protocol capable of dealing with networks with high error rates and as a uniform solution for many different scenarios. It pays a corresponding price for this capability by using expensive error correction mechanisms (resends triggered by timeouts, message numbers and sliding windows for administration). This overhead is wasted in combination with low-error networks. Similarly, TCP's slow start and related mechanisms for flow control are often not needed in a SAN environment. Since a number of SANs are commercially available, they can be regarded as COTS equipment, even though their use requires some change or additions to the networking subsystem of standard operating systems. Therefore, SANs like Myrinet present today's most promising platform for cluster-based parallel computing.

Not only the protocol used over the underlying network is important, but also the way the network can be accessed by applications—solutions to this problem are discussed in the next section.

3.2.2 Accessing network interfaces

In a supercomputer, the problem of accessing the network interface is a simple one. Often, supercomputers are not used in a multiprogrammed mode, but run one program at a time in time-sharing mode, or are space-shared between programs. This removes the need to provide protected and secure access to the network interface. Also, the network interface can be placed directly onto the memory bus (as it is done in, e.g., the T3E [251]), avoiding the expenses of accessing an Input/Output (I/O)-bus.

In a COTS environment, on the other hand, protections are necessary and placement of network interfaces has to comply with standard architectures. Network interfaces can only be placed on the I/O-bus without violating the COTS principle, given today's common system architectures. The time-sharing nature of a COTS environment requires virtualization of the network interface: the operating system makes multiple logical communication endpoints accessible to different programs, maps these endpoints to the single interface and protects access to these endpoints. Accessing such an endpoint then requires an expensive trap into the operating system, plus additional copying of data. On the receiver side, arriving messages cause an interrupt,

and the application has to perform another system call and a copy operation. While this allows very simple network interface hardware, the runtime overhead is considerable.

The network interface cards of, e.g., Myrinet [39], Memory Channel [88] or SHRIMP [38] have additional hardware that allows more intelligent solutions; the Myrinet adapter, e.g., has a proprietary 33 MHz LANai processor on board. The issues to resolve when using such an intelligent network interface are, among others, the question of data transfer between host memory and interface memory, address translation for Direct Memory Access (DMA) transfers, protecting and virtualizing the interface, controlling the time of data transfer (use interrupts or polling?), reliability of communication, fragmentation and reassembly of messages, and multicast. There are a number of design choices for each of these questions, and a number of protocols have been proposed. BHOEDJANG et al. [32] give an excellent overview and ARAKI et al. [11] present experimental results for a number of popular implementations of low-level access methods for network interfaces.[1]

A common property of all these protocols is to allow user-level access to the network by bypassing the operating system kernel. But some of them provide only limited support for multiprogramming and a very crude communication semantics (e.g., unreliable, possibly out-of-order delivery). Aggressive implementations of low-level software layers like Active Messages [294] and, on top of it, Fast Messages [220] provide reasonable communication semantics (like in-order delivery of messages, fault detection[2]) while still delivering almost the entire hardware performance to the application. In particular, end-to-end bandwidth and latency between applications is only little different from what the underlying hardware would be capable of.

Fast Messages and similar systems still lack support for multiprogramming and multiple connections. Such support is provided by additional software layers like U-Net [30, 293], which allows virtualized user-level access to the network interface, or VMMC-2 [72] (similar with some slight differences). These systems also circumvent the necessity of pinning down memory pages for DMA transfer by incorporating a translation look aside buffer access in the protocol processing. Other systems with similar objectives are discussed in [32].

As a consequence of this large number of possible approaches with their own respective advantages and disadvantages, a common standard has not emerged until recently. The Virtual Interface Architecture (VIA) proposal [291], introduced by Intel, Microsoft, and Compaq, targets this problem by proposing a common standard for direct access to virtualized network interfaces and can be expected to have a large impact on COTS systems. VIA is based mostly on the U-Net architecture and adds remote memory references and virtual address handling from VMMC-2. Unlike most other systems, a virtual interface is connected only to exactly one other virtual interface. VIA significantly improves communication performance over standard communication: remote operations in the Component Object Model (COM) over VIA are faster than local operations with standard COM [295].

The problem of how the operating system can make networking resources available to application programs is also addressed by DRUSCHEL [71]: it is argued that poor I/O performance for high-performance networking devices is due to the cache not being able to mask memory latency since in current systems locality of reference is not given for I/O devices. This lack of locality is caused by (1) excessive data copying during message processing (lack of integration of I/O components, protection boundaries), (2) scheduling problems for interrupt handlers, kernel routines and the application itself, which can destroy locality, and (3) the fact that the kernel has to be involved in network processing, which entails costly switching between different protection boundaries. DRUSCHEL [71] suggests the fbufs mechanism to transport data efficiently across protection boundaries and application device channels to give applications direct but controlled access to the network interface, allowing user-level protocol processing (which is similar to, e.g., U-Net). Direct user-level network access allows the implementation of efficient zero-copy protocols.

MUKHERJEE and HILL [205] take a long-term perspective and propose to integrate network interfaces on the memory bus even in COTS systems, lest the network becomes an even more severe performance bottleneck in future systems than it already is today. They suggest to treat the network interface similar to memory and not

[1] E.g., the socket interface on top of Myrinet is about one order of magnitude slower than what the hardware is capable of.

[2] Owing to the very small error rates of, e.g., Myrinet, some systems consider a communication error a fatal event that has to be handled by higher level mechanisms like restart and can therefore forego fault masking techniques like acknowledgement and retransmission.

like a disk device, so that the virtualization can be handled by the virtual memory system. A main argument for this design is the fact that already today SAN link bandwidth has exceeded I/O bus bandwidth and is almost as fast as a typical memory bus (4 Gbps), and it is expected to exceed memory bus speed in the near future.

From all this research, it is possible to draw the following observation: While there is still a gap in communication performance between COTS clusters and custom-designed supercomputers, tremendous progress towards closing this gap has been made. In particular, the introduction of switch-based communication hardware similar to that found in supercomputers has made competitive communication performance available to clusters. Work on virtualized network access and aggressive network layers provides a reasonable percentage of the raw hardware communication performance to an application with a reasonably abstract application programming interface.

3.2.3 Parallel computing

Given fast workstations and fast interconnection networks, how can clusters be used for parallel computing? Several paradigms exist. A first classification can be made according to the programming paradigm: message passing systems or Distributed Shared Memory (DSM) systems.

Message passing

The conceptually simplest means to have two entities communicate with each other is by explicitly exchanging messages between them. One well known theoretical model for this is Communicating Sequential Processes (CSP) [107]. In a message passing system, a programmer has to specify all communication between concurrent parts of a program—which data is to be sent where, received when, etc. The actual communication primitives come in many flavors, e.g., synchronized, unsynchronized, or with a procedure call semantic.

Several message passing systems have been introduced that more or less follow this approach (see [195] for an overview). Two of the most popular message passing systems are Parallel Virtual Machine (PVM) [279] and Message Passing Interface (MPI) [70, 199, 200]. MPI, due to its support by many computer and software vendors, can be expected to be the message passing standard of the foreseeable future.

Distributed shared memory

When writing a message passing program, a programmer has to specify exactly which data has to be moved between which processors at what time to exchange information between processors. This is often considered tedious. A much simpler model is a single address space that is shared among all the processors, which execute independent programs—the theoretical model is the Parallel Random Access Machine (PRAM) [81, 102]. DSM systems implement this model for multiple processors that do not have any physical memory in common; an overview can be found in, e.g., [233]. Message passing systems are commonly considered to allow better low-level fine tuning of a parallel program, whereas DSM lends itself to a simpler programming style without loosing too much efficiency if aggressively implemented. However, these aggressive implementations carry the burden of complicated consistency protocols that potentially lessen the ease of programming so that a compromise for this tradeoff has to be found. In Section 9.2, this question of a semantic gap between message passing and distributed shared memory is reconsidered.

Implementing DSM semantics is faced with a number of problems. One such problem is the need for a model of the shared memory's semantics under read and write accesses. Such a model is called the memory consistency semantics[3] [2, 215] and can be considered as a contract between the programmer and the memory system.

[3] There is quite a bit of confusion in the literature about the usage of "memory consistency" and "coherency". Some writers make a distinction between coherence as the general term for memory semantics, and consistency as any specific kind of coherence [215], others use the terms interchangeably [2]. Since this distinction is not important in the context of this dissertation, the latter approach is followed.

The need for such a semantics arises since it is, in a distributed memory setting, no longer clear how read and write accesses to memory are perceived by the various processors. Intuitively, a programmer would expect a read to return the value most recently written to that location (which is called strict consistency). However, in a distributed setting, "most recently" is not clearly defined. A well defined semantics is "sequential consistency", where any interleaving of operations from all the processors' programs is a valid execution, but no other [157]. While straightforward to use, sequential consistency is considered to result in a slow program execution. Hence, weaker or relaxed consistency models were proposed and used in various implementations of DSM; examples are discussed below.

Recently, the use of such relaxed memory models has been revisited in the light of new developments in processor technology. The reason for introducing relaxed models is their ability to use additional optimizations in the protocol that implements the memory consistency protocol. The simpler and more intuitive sequential consistency model [157] has been criticized for not allowing such optimizations. Consequently, most implementations of shared memory systems have used relaxed or weak memory consistency of one form or another. However, standard microprocessor architecture has developed (among other things) the concept of speculative execution. HILL [105] shows that with speculative execution the performance gain of weak consistency models as compared to aggressive implementations of sequential consistency is only about 20% in the best case and even smaller in most other cases. Yet the complicated user interface of weak models remains. HILL argues convincingly that this small benefit does not merit the additional cost to software developers.

In addition to the consistency model, DSM systems can be classified according to how the shared memory is implemented: in hardware or in software. Examples for hardware DSM systems include the Stanford Dash [173], the MIT Alewife [3], or the newer SGI Origin [164] machines. For the use in a cluster, software DSM solutions are more relevant.

One of the first software DSM systems is IVY, introduced by LI [175, 177]. IVY, like many following systems, uses the virtual memory hardware to intercept access to the shared memory region and uses the operating system's page fault handler for this intercepted access to ensure the memory's consistency, e.g., by fetching memory contents that is not locally available from another processor. The granularity with which memory is managed is the virtual memory system's page size. IVY's performance suffers from the strict consistency model it implements. Treadmarks [7], one of the more successful later projects that use the virtual memory approach, overcomes this problem by implementing lazy release consistency (access to shared memory must be explicitly announced with acquire and release operations [215]) LU et al. [184] compare the performance of a number of benchmark applications when implemented with TreadMarks or with PVM. The performance is found to be comparable, but programming is much easier with the DSM system, in particular for larger programs.

Using virtual memory pages as unit of sharing opens the possibility of false sharing: unrelated variables can be on the same page, and this page is frequently moved back and forth between multiple processors. To avoid this problem, individual variables or objects can be used directly as units of sharing. An example for such an approach is Munin [47] where the programmer gives hints on the expected access pattern of shared variables (e.g., "producer-consumer") and the system chooses a suitable protocol to implement release consistency for this pattern. Midway [31] extends Munin by introducing entry consistency, where synchronization objects and data objects are explicitly coupled and a program can request that only individual data objects are made consistent, not the entire shared memory as is the case for release consistency.

Munin and Midway share individual variables. This approach can be taken a step further if object-based systems are used. One example is Orca [18, 19]: Shared data is encapsulated in shared objects, which can only be accessed through corresponding methods. Such a data access includes synchronization operations and results in a sequentially consistent memory model. In Orca, the compiler generates information about the usage pattern of shared objects that is then used by the runtime system to implement efficient data distribution. Unlike Orca, which defines its own language, Shared Objects Network-interconnected Computer (SONiC) [227, 228] uses C++ and allows a programmer to implement his own consistency models by extending supplied data classes. Again, data access via object methods is used to enforce consistency. Aurora [185] is similar but additionally allows the consistency model of an object to be changed during execution, so that in different phases of a program optimal consistency protocols can be used. A slightly different ap-

proach to programming a DSM system is taken by Linda [35, 46]: an explicit tuplespace is defined to/from which data can be written or read.

Two examples for lower-level approaches to DSM are the C Region Library (CRL) [120] and Cid [212]. CRL is a library of C functions that implement a DSM system. The source code must be annotated with calls explicitly mapping shared data into local memory. Cid is quite similar to CRL, extending C with source code annotations to identify global objects. Cid is more general than CRL owing to better multithreading support and the programmer's ability to influence data placement. These systems show how with very modest support from an underlying runtime environment DSM semantics can be implemented, albeit not transparently.

Jade [241, 242] is an interesting case of DSM without explicit parallelism: Units of code are identified as tasks and the programmer provides data access information for these tasks; Jade's runtime system dynamically extracts the parallel execution from this description.

Some projects have considered problems of fault tolerance in a DSM setting. Handling faults is important in any kind of parallel architecture (owing to the large number of elements that can reduce the system's reliability, but which are also an opportunity for redundancy). This is even more so the case in a cluster environment without uniform system administration: an individual workstation can be rebooted far easier than a parallel supercomputer.

One example for such a fault-tolerant DSM system is Calypso [23, 64]. Calypso is interesting for its integrated handling of load balancing and fault masking. Since it uses a quite simple yet powerful execution model, it serves as a basis for the discussion of responsive execution of parallel programs in Chapter 4, which also contains a more detailed description of Calypso.

JEONG et al. [118] extend the Linda model to allow recovery from failed processes by checkpointing the tuplespace and investigate the tradeoffs between recovery time and performance in the failure-free case for a number of different mechanisms. Checkpointing in general has been a popular mechanism to achieve fault tolerance in DSM systems; other examples based on checkpointing include DOME [10], which is additionally interesting since it not only considers problems of load balancing among CPUs but also among network resources, and ICARE [143], which occasionally synchronizes the nodes in a DSM system, takes a globally consistent checkpoint, and stores the checkpoint information redundantly in the main memory of the nodes (avoiding the overhead of storing a checkpoint on disks). Checkpointing is also employed as a mechanism to improve the responsiveness of a Calypso program in Chapter 6.

Despite all this research, software DSM systems have so far failed to make a big impact on mainstream computing, whereas Symmetric Multiprocessing (SMP) systems have.[4] This difference can be mainly attributed to the higher performance of SMP systems and the higher coding overhead for DSM systems. Shasta [248] is an attempt to leverage existing applications for SMP machines and execute them in a clustered DSM environment only by rewriting the object code. Shasta checks every read and write against memory to ensure that data is available locally (similar to some objects-based systems like, e.g., SONiC), but heavily optimizes this check so that the overhead is only about 20 % compared with an unchecked execution. Additionally, Shasta ensures that the application can transparently call the operating system, no matter where a process is located. Shasta is even able to execute the Oracle 7.3 database engine distributedly without needing its source code.

3.2.4 Metacomputing

Metacomputing [263] is the attempt to make use of geographically distributed resources to tackle emerging high-performance applications. Apart from the classical supercomputers or clusters of workstations providing computational power, resources can also mean large databases, advanced visualization equipment, or even scientific instruments like telescopes. GRIMSHAW et al. [96] identify shared persistent object spaces, transparent remote execution, wide area queueing systems and parallel processing as building blocks for true meta-applications. Such meta-applications consist of multiple, often independently developed applications that

[4] In an SMP, a number of processors share some physical memory and the time to access any memory location is usually constant for all processors—hence the name "symmetric."

each require vast amounts of resources. An example for such an application is a coupled ocean/atmosphere simulation [96]. A number of systems have been developed that address some of metacomputing's challenges.

One such system is Charlotte [27], a reimplementation of the main Calypso ideas in Java. On the basis of Charlotte, some issues that arise in wide area parallel computing are discussed in Chapter 9. The Gallop system [300] addresses wide area scheduling for sites distributed over the Internet. Experiments show that even with today's limited Internet bandwidth, performance benefits can be achieved with parallel processing over the Internet.

GRIMSHAW and WULF [97] describe the Legion system. Legion's aim is a single, coherent, virtual machine that scales to large configurations of real systems, is easy to program, tolerates faults, accommodates heterogeneity, and provides security guarantees to both resource providers and users.

Globus [82] is a system that tries to build a practical metacomputing environment. FOSTER and KESSELMAN [82] argue that, while metacomputing shares some commonalities with traditional distributed and parallel platforms, additional research is needed, e.g., in the context of resource discovery or scheduling of computation and/or communication. Applications are supposed to be able to adapt to the execution environment, which is often discovered only at runtime. Globus identifies four main application areas for metacomputing: (1) coupling desktop graphics equipments with remote supercomputers, (2) combining instruments (e.g., satellites) with remote machines to allow real-time visualization or steering, (3) collaborative computing, and (4) distributed (super-)computing. Obviously, the borders are not strict between these areas.

The inherent unpredictability of wide area environments is in conflict with even minimum Quality-of-Service guarantees. Hence, applications need to obtain appropriate configuration information and have to adapt to it, if possible in real time. To this end, Globus provides resource location, allocation and unified information services. Additionally, the Qualis architecture [167] is the Quality-of-Service component for Globus. It relies on an operating system with Portable Operating System Interface (POSIX)-compatible real-time extensions and on RSVP [42] to guarantee network Quality of Service (the use of Differentiated Services [36] is planned). The application can specify its resource requirements directly, or, for supercomputer applications, historical information can be used to predict future runtimes [264]. As an example for applications with Quality-of-Service requirements, satellite image processing [166] is described.

3.3 Focus on fault tolerance

Dealing with the inherent imperfections of any real hardware or software is one of the original problems of fault tolerance. A real component is always susceptible to failure, which endangers the correct functioning of a system as a whole. System failures are often unacceptable, given the immense importance of today's computer applications where large amounts of money or even lives can depend on the correct functioning of these applications (examples can be found in [85, 101]). Therefore, mechanisms to handle such failures are necessary. And as GRAY and SIEWIOREK [95] point out, the larger a system is, the more important is its high availability—but also the less likely it is to actually be highly available owing to its sheer size and complexity.

Devising such fault-tolerance mechanisms requires research in a number of different areas, which are briefly discussed in the remainder of this section, along with a number of examples—overviews and introductions can be found in, e.g, [8, 62, 95, 231, 259, 260, 280].

The basis for fault tolerance is formed by precise models of a system and the possible faults it can experience. A fundamental issue is the distinction between a fault, which is the ultimate cause of any misfunction but only creates the latent potential for it; an error, which is an undesired circumstance internal to the system and caused by a fault; and finally a failure, which is a user-observed deviation of the system from its specification, caused by (one or more) errors [159, 160]. Sometimes, only faults and failures are considered [148]. Faults, errors, or failures, can then be further classified according to their nature, e.g., a machine can crash or a program can compute wrong results. Another classification of faults is according to the time a fault occured: e.g., when designing a system or during its deployment. A number of such fault models have been proposed (cp., e.g., [62, 161, 163, 229]). Also important is the level where a fault occurs: hardware, operating system, application programs, and so on. Today, hardware is a relatively minor cause of system failures when

compared to software or environment-related causes [95].

Models of a system's behavior can also be used to evaluate parameters of a system: reliability evaluation is an example [40], where reliability at time t is the probability that a system works correctly in the entire interval of time $[0, t]$, provided it worked correctly at time 0. Typical assumptions for such model evaluations are the reliability of individual components and stochastic properties like independence of faults. An analysis somewhat similar to reliability analysis is undertaken in Section 5 for some aspects of the Calypso system.

A basic technique to compensate for potential errors is additional redundancy, either in space or time. Redundancy in space can mean, e.g., duplicating processing elements, redundancy in time can be achieved by, e.g., repeatedly executing an algorithm on one machine and comparing the results of the executions. Usually, neither of these kinds of redundancy appears in isolation and it can be applied at different levels of a system. Finally, different forms of redundancy, like hot or cold standby, can be distinguished.

Another characteristic of fault-tolerance mechanisms is whether they are active before or after an error occurs: forward error recovery tries to deal with errors even before they happen, while backward error recovery only takes minimal steps before error occurrence and repairs the damage afterwards. Again, both mechanisms are usually mixed to some degree.

On the basis of such models and techniques, algorithms can be developed that tolerate faults. A typical class of such algorithms are consensus-based algorithms [28], where a number of active entities, e.g., processors, have to reach a common accord on some data items, even in the presence of faulty or malicious processors. Consensus-based frameworks for fault tolerance have been proposed [190, 191], and consensus is at the center of the CORE system, described more closely in Section 3.5.4.

Linked with the question of algorithms is that of data representation. Again, the principle of redundancy can be efficiently used by representing data with a coding that has enough information to protect against faults. Perhaps the simplest example is parity: for each data word, a single bit is additionally stored that encodes whether the number of set bits in this word is odd or even. Parity is generalized by Cyclic Redundancy Check (CRC), which adds a few check bits to a message and allows detection of corrupted messages with low overhead (easily implemented in hardware) and very high probability.

Detecting errors is not always as easy as with CRC; it highly depends on the desired fault model. Supervising processors with additional watchdog hardware or software is one example; acceptance, timing or plausibility checks are others [8]. In a distributed system, detection can be done by multiple machines, diagnosing each other. Here the problem occurs how to diagnose which processors are actually faulty when contradictory diagnosis results are found—this has first been addressed by PREPARATA et al. [232] (see [194] for an overview of system diagnosis). Somewhat similar problems arise in testing a system before deployment.

After an error has been detected (and, if necessary, properly diagnosed), some actions must be taken to compensate for it. One such measure is rollback recovery, where the system returns to a previous state that is considered to be correct. Checkpointing is a way to implement such a rollback step: the system periodically writes state information to stable storage and, upon detecting an error, reads in such a checkpoint and resumes processing from this point onwards, effectively retrying the previous execution sequence. One problem here is how to choose the interval of writing checkpoints so as to optimize a desired metric. In Chapter 6, an analysis of the checkpointing interval problem for optimizing responsiveness is presented, along with additional related work.

All these techniques and mechanisms together have one main objective: to ensure failure-free, continuous service of a system. But no single mechanism can achieve this in isolation; they have to be integrated in a complete system design. The following sections discuss a few exemplary systems, as well as some paradigms mentioned above in more detail.

3.3.1 Custom-build systems

Software Implemented Fault Tolerance—SIFT

Software Implemented Fault Tolerance (SIFT) [299] is an operating system designed for use in flight-critical functions in commercial aircrafts. Such applications require a very high reliability. SIFT's approach is to

use standard, simple hardware and implement most of the fault tolerance in software. The basic design is a number of star-connected processors, where the interconnection network is used to broadcast messages to all peer processors. SIFT provides services such as scheduling, synchronization, consistency, communications, fault masking and reconfiguration.

The main abstraction in SIFT is the task, the unit of computation. Tasks are scheduled at precomputed times, uniformly on all processors. Data produced by tasks is broadcast to all other processors, and then a special voter task compares the results of an application task using majority voting. Processors with deviating results are marked as faulty, and a reconfiguration is initiated to exclude such a processor from further processing.

This software-based voting permits a much looser clock synchronization than needed for hardware-implemented voting schemes (by about three orders of magnitude). SIFT proposed a novel clock synchronization mechanism that achieves the desired precision while being able to mask faulty clocks. One goal of SIFT was to simplify the correctness proofs of the design of a fault-tolerant system. However, the overhead introduced by software fault tolerance turned out to be very high: SIFT was found to use up to 60% of processing time for internal functions [280, p. 397].

Error Resistant Interactively Consistent Architecture—ERICA

Unlike SIFT, the Error Resistant Interactively Consistent Architecture (ERICA) [288] concentrates on hardware solutions to build a computer that behaves correctly despite hardware failures. In ERICA, a processor/memory module is replaced by n processors and k times the memory. The memory is spread over the n processors, so that one module now has k/n times the original amount of memory. Access to memory happens via a special encoder/decoder logic: using an appropriate (n, k) code, memory content is stored redundantly. Given large enough values of n and k, failures of entire modules can be tolerated—this is the case for $n = 4$ and $k = 2$. Since the hardware hides the redundant memory and replicated processors from the software, only minimal modifications to operating system or applications are necessary. Indeed, an operating system developed for a non-redundant machine was used unaltered on a redundant version. It is a particular strength of ERICA that any architecture can be systematically transformed in a redundant counterpart, provided the hardware behaves deterministically.

The values of n and k determine system reliability and cost. Reliability improvement is here measured as the ratio between mean time to repair of the redundant and the non-redundant system, cost ratio is defined analogously. Typical approaches like double redundancy ($n = k = 2$) or triple modular redundancy ($n = 3$, $k = 1$) are shown to deliver only modest reliability improvements [288]. In ERICA, the $(4, 2)$-concept is used, which provides high reliability improvements at costs comparable to triple modular redundancy. Other designs exceed the reliability of a $(4, 2)$ system, but pay a higher premium in cost. Additionally, four processors is the smallest possible number for which the Byzantine generals problem can be solved; in ERICA, a special chip implements a protocol for solving it.

The particular strength of ERICA is the transparent implementation of fault tolerance, combined with a systematic approach to construct a redundant system architecture. Theoretical results concerning (n, k)-coding strengthen the case made for ERICA.

3.3.2 Group communication

One popular approach to fault tolerance is to use replicated entities or objects. Replication often results in a consistency problem, and the process group approach has been suggested [34] to address it with a simple programming interface by providing a single-object view on a group of objects.

The abstract idea of the process group approach is implemented by group communication protocols. To do so, a group communication protocol has to solve a number of problems, namely providing consistent semantics of message delivery and the management of group membership. The consistency of a group communication protocol is characterized by the messages order characteristics that it guarantees. A clear definition of such message orders is necessitated by the nature of many-to-many communication. For example, two messages

sent by two senders can be observed in different orders by two receivers. Depending on the application semantics, this may or may not be acceptable. A group communication protocol ensures uniform behavior in such cases with a number of possible guarantee levels: Reliable message delivery is usually required from a group communication protocol, it could guarantee that messages are always received in the order they were sent (FIFO ordering); it could also guarantee that all potential causal dependencies between messages are respected by message receivers (causal ordering); or all processors could deliver all messages in exactly the same order (total order property).

In the presence of node failures, the notion of reliable delivery is also somewhat more complicated than it is the case for unicast communication. Additionally, group communication protocols used for fault tolerance should be fault-tolerant themselves. Many protocols have been designed to cope with different fault scenarios; an overview of issues in group communication can be found, e.g., in [60, 99].

One well known example of a group communication system is ISIS [34], later redesigned as Horus [289]. Other systems include the global sequencer [124], a 3-phase commit protocol [186], Transis [67], or the Totem protocol [204], which are discussed in more detail in Section 7.4.

With regard to responsiveness, the behavior of group communication protocols in real time is of interest. While many protocols have a well defined logical semantics, even in the presence of faults, only little attention has been paid to real-time aspects. An experimental investigation of this question for the Totem protocol is presented in Section 7.4.

3.3.3 Cluster-based availability

A number of projects aim at using clusters to provide increased availability for services (PFISTER [223] gives an overview). The prospect is a tempting one indeed: where traditional commercial fault-tolerance architectures (such as Tandem's Integrity systems [119]) have struggled to incorporate redundancy in a single machine, today machines are cheap enough to use an entire system as a unit of redundancy—difficult hardware questions like hot-pluggable CPUs are a non-issue when an entire machine can be plugged in and out, with software taking care of consistency. In a similar vein are systems that attempt to present a cluster as a single machine; examples include Solaris MC [144, 256] or the single system image proposal for UnixWare from Compaq [297]. However, these systems deal with failures only superficially (e.g., while Solaris MC survives the crash of any machine, processes running on this machine are simply lost).

Hence, there remain open questions. Consistency is one of them, efficiency another. Transparency to clients is another obvious necessity. A few cluster-based projects aiming at increased service availability are discussed in the following sections.

SunSCALR

SINGHAI et al. [262] propose SunSCALR, a design for a highly available, scalable, and inexpensive server for internet-related services such as the WWW. The design consists of a cluster of workstations with standard UNIX operating systems. A specific service, such as a WWW server, is associated with a group of machines from this cluster. If any machine in this cluster fails (fail-stop behavior is assumed), its IP address is assigned to another machine (selected with a leader election protocol, peer hosts and routers are informed of the new location of this IP address). The service is then restarted on this new host (similarly if only a service, but not the entire machine fails). This is called IP failover and is the core mechanism for scalability and availability.

Failures are detected with heartbeat messages: every host cyclically broadcast an alive message; if more than a certain number of these messages are lost, a host is deemed to have failed. Additionally, this heartbeat message can also be used to communicate load information and balance the load of individual servers by temporarily reassigning IP addresses. IP failover also allows a simple integration of additional or repaired machines, implying on-line scalability.

This scheme allows a very simple and efficient implementation (failover latency in SunSCALR is around 10 s [262]) of a highly available distributed server for many applications. Since it is IP based (and not based on distributed name servers like some other proposals with similar intent), even clients that cache server address

mappings do not observe the failure of a server machine in between service requests. However, this is only true if the application is stateless (like, e.g., WWW) or is capable to handle restarted servers. This is usually the case if an application can reissue service requests multiple times without changing the semantics, i.e., if it is idempotent. Hence, SunSCALR is not fully transparent, but closely matches important applications from the Internet context.

Wolfpack

The Microsoft Corp. recently introduced a clustering extension to their popular Windows NT operating system called "Windows NT Clustering Service" [85, 257, 292], also known as "Wolfpack". Main concern of Wolfpack is to improve availability of servers during hardware and/or software failure. Other goals are increased scalability—which is somewhat debatable considering that in the original Wolfpack version only two nodes can be used—and better management functionality. Applications that use such a server are presented with the illusion of a single, powerful, and highly available machine.

Wolfpack uses four abstractions to structure its approach: nodes, resources, resource dependencies, and resource groups. A resource is the basic unit of management like a disk or an IP address. Resources can be bundled into logical groups that are managed as a single entity and also form the unit of migration between nodes.

Wolfpack clusters are based on the "shared nothing" principle: Any resource available in the cluster is owned by exactly one node. In case of failure of this node, the clustering software detects this failure (by means of a simple heartbeat mechanism) and then moves all resources owned by this node to another system in the cluster ("failover"). A software resource has to be restarted by the cluster service. Resources can also be explicitly pulled from or pushed to some nodes. However, such a migration of working resources results in temporary service outage.

The existence of such service outages (either due to voluntary migration or failover) implies that the access to cluster-based resources is not completely transparent for clients that do have a state: they must reconnect to this server after failover has been completed. For such applications, Wolfpack provides a semantics similar to that of monolithic systems that employ restart mechanisms. This situation is even less convenient with intermediary software layers (e.g., database engines) that transparently perform the reconnection, but loose application state.

Comparing Wolfpack with SunSCALR shows that the simpler mechanisms of the latter do not necessarily impede its functionality. SunSCALR's IP-based mechanisms give it the same level of transparency and better scalability than Wolfpack's somewhat complicated architecture. Also, Wolfpack is tightly coupled to one particular operating system. Both systems employ a cold standby approach and have to restart software services after failures are detected. It remains to be seen how the development of Wolfpack will proceed.

NonStop Cluster Application Protection System—NCAPS

Tandem's NonStop Cluster Application Protection System (NCAPS) [162] shall serve as a last example for cluster-based high availability solutions. NCAPS leverages the high performance of a ServerNet-based fault-tolerant architecture [17] to build a simple programming environment for improving the availability of applications running on UNIX clusters. Unlike Wolfpack, it uses a warm standby approach: an application process is accompanied by a backup process (running on a different node) that is in an idle state and takes over when the primary process fails. That backup process does not provide service, it is only initialized. The backup process allows very fast failover (on the order of 10 s), as opposed to Wolfpack's restart approach (application start and initialization can take up to 20 minutes in an example in [162]).

To implement this, a number of small services is used: a heartbeat-based node monitor, a keep-alive service that restarts failed applications, and, at NCAPS's core, a Process Pair Manager (PPM). This PPM is responsible for detecting application failure, promoting a backup process to primary status, and starting another backup. An application has to be linked with a special library to work with the PPM.

This last requirement is also NCAPS largest impediment: it is not completely suitable to legacy systems. Therefore, there is still an open issue: fast, transparent failover. A possible solution for this problem that employs hot standby and can flexibly adapt to different fault models is described in Section 7.

3.4 Focus on real time

What is a real-time system? Evidently a system for that, somehow, real, physical time is of importance, that must function in real, actual time and not in a virtual time as so often used in computer science. One definition states that "a real-time system [is] a system that changes its state as a function of real time" [152]. In a strict interpretation, this is true of any system. Additional insight can be gained from STANKOVIC: "In real-time computing the correctness of the system depends not only on the logical result of the computation but also on the time at which the results are produced" [271]. Usually this means that a result is only useful if it is produced before or by a given time; this time is commonly called a deadline. A real-time system is then characterized as a system that guarantees to meet deadlines for a certain set of services. To be able to give this guarantee, usually a considerable amount of assumptions concerning the system itself, the load, possible faults, etc. are necessary.

The definition of real-time systems is somewhat blurred by the commonly found distinction between hard and soft real-time systems. Hard real-time systems can be considered to be systems where the cost of timing failures can be orders of magnitude higher than the normal utility of a system [152], and a soft real-time system as one where cost of failure and utility are comparable. But this is a non-operational definition. Another common definition states that a real-time system is hard if all deadlines must be met (for whatever reason), and soft if occasional failure to meet a deadline is tolerable (cp., e.g., [153]). This definition separates specification requirements from the motivation for these requirements and is therefore preferable. However, it is often the case that deadlines must be met for the very reason that failures have unacceptable consequences. The motivation for considering hard real-time systems can be found in many application scenarios: process control (in one form or another) is a typical one. For a computer controlling a physical process, there is no way to influence the passing of time and external events, which is dictated by the laws of physics. Therefore, a computation must be finished by deadline or the control over a physical might be lost. Since many such process control applications indeed involve critical and often dangerous systems, the concentration on absolutely guaranteed deadlines is only natural. But as the discussion in Chapter 2 has pointed out, it is ultimately unrealistic to attempt to build an actual hard real-time system with 100% guarantees; only probabilistic approaches have any hope of properly reflecting reality. Therefore, hardness and softness of real-time systems are only points on a continuous scale and not rigid opposites. But given the vagueness of a number of definitions of real-time systems and the possibility of circular definitions, KRISHNA and SHIN conclude after a discussion of possible definitions for real-time systems: "A real-time system is anything that we, the authors of this book, consider to be a real-time system" [153, p. 2]. Further discussions about the nature of real-time systems can be found in introductions to the field [149, 153, 270, 271, 274].

Traditional research in real time has mostly considered hard real-time systems. Typical models consider tasks (with fixed deadlines) that can be periodic, aperiodic (arriving at any time) or sporadic (arriving at any time but with minimal inter-arrival interval). Scheduling research has produced a large amount of results regarding such task models (see [276] for an overview); typical examples are the Earliest Deadline First (EDF) algorithm or the Rate Monotonic Scheduling (RMS) algorithm, both described in [182]. EDF is a dynamic priority algorithm for preemptable tasks, the task with the shortest deadline has the highest priority. As a typical result, it is possible to show that as long as the utilization (the ratio of execution time and deadline) of all tasks together is smaller than 1, EDF will be able to find a schedule [182]. In RMS, tasks are assumed to be preemptable and periodic and the deadline equals the period. Using RMS, a set of n tasks is always schedulable if the total utilization does not exceed $n(2^{1/n} - 1)$ [171]. Such scheduling algorithms have to be implemented by proper operating system environments. Real-time kernels are operating systems that try to deal with the problem of executing such scheduling algorithms on a real system. A new challenge for such systems is to combine flexibility with predictability: operate in a dynamic environment while predicting or

even resolving resource conflicts to ensure timing constraints [270].

With the advent of distributed multimedia, true examples for soft real-time systems have appeared. At the core of multimedia applications (such as teleconferencing or even video games) is real-time technology, but without the stringent reliability requirements often found in classical real-time applications. However, even in multimedia applications, hard real-time requirements exist: audio playback quality, e.g., degrades considerably if deadlines are missed, whereas video playback is quite tolerant [121]. An integration of hard real-time applications like factory floor automation with integrated multimedia applications is a challenging vision for the future [270], in particular in the context of COTS environments.

In recent years, there has been some standardization efforts with regard to real-time systems. These efforts include proposals for operating systems (the POSIX standards [113]) and real-time object management [218]; real-time communication is discussed in Section 3.6.1. Some examples for real-time systems are discussed in more detail in the following sections.

3.4.1 Spring

The Spring kernel [273] is an example for a real-time operating system designed to handle hard real-time requirements. Its ultimate objectives are application-level predictability combined with flexibility in large, complex, and evolving real-time environments. To do so, Spring uses a number of innovative mechanisms.

The basic abstraction in Spring is a task, which is annotated with both importance and timing requirements. From these two attributes, three classes of tasks are formed: critical tasks must meet their deadlines under all circumstances (subject to an assumed fault model); essential tasks are necessary to the system's correct operation and do have timing requirements, but do not cause catastrophe if they are not completed on time; and unessential tasks. There are typically only a few critical tasks, but many essential tasks with varying levels of importance.

The stringent requirements of critical tasks predicate the need for a priori guarantees on their execution. For essential tasks, however, such guarantees are not feasible owing to their large number. Spring deals with this problem by guaranteeing essential tasks on-line by means of an admission test and introduces the notion of a currently guaranteed task set: the set of tasks for which on-line scheduling guarantees fulfilling the timing requirements (it is not always possible to guarantee all tasks in a dynamic environment). It is therefore globally known in a Spring system which tasks will succeed, or which might only possibly succeed. This is in sharp contrast to probabilistic approaches that only give success probabilities; it also allows an unguaranteed task, e.g., to initiate error-handling activities immediately upon arrival.

Spring is implemented on a set of interconnected nodes, where each node consists of a number of application processors, a system processor, and an I/O subsystem. The application processors perform the application tasks, the system processor deals with operating system functionality such as scheduling and handling of interrupts from the I/O subsystem (hiding interrupts from application processors increases predictability of task runtimes). All processors run a low-level, table-driven dispatcher. The system processor additionally runs a local scheduler that performs the admission test for guaranteeing essential tasks and computes the dispatcher table, a distributed scheduler that attempts to find a node for tasks that could not be guaranteed locally, and a meta-level controller that switches general policies or parameters according to changes in the environment. Spring's scheduler is also able to handle entire groups of tasks with a single deadline.

Spring's goal of predictability combined with flexibility is ambitious. Spring does achieve a great deal and presents a working system. However, there is still a number of unresolved questions with regard to, e.g., design rules that facilitate later implementation and analysis, languages that support it, and fast yet predictable hardware implementations.

3.4.2 Rialto

Rialto [121, 122] is a proposal for a real-time architecture that is concerned with other application scenarios than Spring. Rialto is designed to support coexisting, independently authored real-time and non-real-time programs and dynamically arbitrates their resource requirements. This approach is quite different from traditional

hard real-time systems where a timing and resource analysis can be done statically before runtime, since the set of tasks is fixed.

The Rialto system uses a simple programming model to express timing requirements: an activity, enclosed in `BeginConstraint`/`EndConstraint` pairs. `BeginConstraint` allows an activity to announce its estimated resource requirements, an attached deadline and a criticality; `EndConstraint` returns the amount of actually consumed resources. The idea here is to use the `EndConstraint` results from previous executions to improve the estimation. Additionally, `BeginConstraint` returns information whether or not this activity is currently schedulable to terminate before its desired deadline, allowing an application to proactively shed load and avoid overload conditions. It is interesting to note that this API does not have a concept of priorities, which are judged to be difficult to arbitrate when independently developed applications are to interact [121].

These constraints are implemented by a modified minimum laxity first scheduler with hysteresis and capacity reserves (similar to [198]). A resource planner, which is similar to the QoS Broker [211] described later, arbitrates between conflicting reservation requests. Laxity-based scheduling, as opposed to earliest deadline scheduling, allows scheduling information to be transmitted in a distributed environment.

A rather interesting property of Rialto is its treatment of clocks. Usual timesharing schedulers handle the clock as a periodic device where preemption only takes place at a clock tic (often 10 or 20 ms). This is not the case in Rialto, where clock tics can be freely programmed; two threads with 300 μs periodic deadlines can be successfully scheduled on a Pentium 90-based system. Some attempts have been made to implement such freely programmable timer interrupts in UNIX-like operating systems (e.g., Linux [21, 269]), but this is still experimental. Similar problems with timer control hamper Windows NT's suitability for real-time tasks with millisecond time scales [123].

Rialto has shown that with a custom-designed kernel, sub-millisecond granularities are feasible even on COTS hardware, and that dynamic time constraints in combination with resource management allow the interaction and coexistence of independently developed, mutually unaware soft real-time and non-real-time applications.

3.4.3 MPI/RT

An important standardization effort to provide real-time capabilities to applications with high-performance demands is MPI/RT [127, 128]. MPI/RT is a proposal for a middleware-implemented API. It emphasizes changes necessary to MPI to make it suitable for embedded, fault-tolerant systems with high performance needs and suggests an interface to define Quality of Service and timeliness requirements for data transfers.

The philosophy of MPI/RT is that while an application programmer knows the Quality-of-Service requirements of the application, he does not necessarily know how to implement them on a given platform. An implementor of the MPI/RT middleware layer, on the other hand, can by virtue of detailed knowledge of the underlaying system (hardware, operating system, runtime system), provide suitable mechanisms to realize such Quality-of-Service requirements. By this separation of concerns, MPI/RT tries to provide portable predictability. However, this separation assumes that it is indeed possible to build such MPI/RT implementations that are capable of providing these guarantees—a question on which MPI/RT explicitly makes no statements since this is of no concern to a specification standard.[5]

The Quality-of-Service requirements are specified as attributes of communication channels (as used by MPI). MPI/RT defines operations to set up, modify and tear down such channels. As real-time programming models, MPI/RT specifies the time-driven, event-driven and priority-driven paradigms. An interesting aspect of the time-driven paradigm is the possibility to specify periodic data transmissions, which are then carried out by the middleware layer without explicit invocations. Overall, MPI/RT provides a very rich expressibility of Quality-of-Service requirements of a program.

This richness, however, can be both a strength and a weakness. For example, it takes eight steps to setup a Quality-of-Service-capable channel between a single sender and receiver [128]. It is questionable in how far all this expressibility on a rather low level of abstraction—the movement of data—is relevant to an application

[5] KANEVSKY et al. [127] mention that "advice to implementors" is a goal of MPI/RT.

programmer, to whom only the timely and correct execution of the program is relevant, and not how this is achieved by the middleware. For some applications such a fine level of control is certainly necessary. Yet it could be argued that it is actually the purpose of a middleware to hide as much details from the application programmer as possible. A comparable dichotomy exists between message passing and distributed shared memory in general. In this dissertation, in contrast to MPI/RT, an approach based on DSM and hiding of details is advocated.

This rather low abstraction level of MPI/RT is also apparent in its fault model. While the necessity of fault tolerance for real-time applications is acknowledged, only limited functionality for fault handling is provided. MPI/RT only allows applications to recover from faults (namely, Quality-of-Service violations) by means of timeout-triggered error handlers, but not to prevent faults. Moreover, the standard suggests to put the burden of specifying these timeouts entirely on the application—which should take into account not only its own requirements, but also details of the platform (Section 4.1 in [127]). But considering specifics of a platform is in conflict with the initial goal of portable predictability. It remains to be seen in how far MPI/RT will have any actual impact. The complexity of the programming model and standard description are not necessarily an asset to MPI/RT.

3.5 Focus on responsiveness

Guaranteeing timing constraints to avoid catastrophic consequences in a real-time system is by itself insufficient if it is not complemented by fault-tolerance mechanisms. It is useless to make sure that all resources are available and that all tasks finish execution in time if the system has failed because of hardware problems. This necessity to combine fault tolerance and real time is addressed by responsive systems. A few of these systems are discussed in some more detail here.

3.5.1 Delta-4

Delta-4 [230] is a research project that seeks to build a distributed, open, fault-tolerant, and real-time architecture. It uses a software implementation for fault tolerance somewhat similar to the SIFT approach (see Section 3.3.1), but is much more encompassing. In Delta-4, a set of (possibly heterogeneous) hosts is connected by a dependable communication system. Hosts are the unit of fault tolerance, and the redundancy necessary for fault tolerance is achieved by replicating software components. The interactions of these replicas is protected from faults via error processing, and failed hosts can be excluded from further participation.

The fault model of Delta-4 distinguishes between fail-silent and fail-uncontrolled hosts. Fail-silent hosts are handled via active or passive replication of software components. Replicas are active if more than one software component has the same state and processes the same input simultaneously; Delta-4 system software chooses among the outputs of these replicas. Passive replicas are merely checkpoints than can be restarted on other hosts if the original host should fail.

Fail-uncontrolled hosts are more difficult to handle. In particular, errors in the time and value domain are possible in this case. For timing errors, the delay over the (shared) communication medium in Delta-4 is unbounded, as is the relative speed and behavior of schedulers on the individual hosts. This asynchrony implies that in Delta-4 only timeouts can be used to handle time-domain errors. A timeout does not necessarily incriminate a late host as faulty, but causes errors at higher application levels. Delta-4 takes a proactive approach of fault avoidance by judiciously configuring the system with regard to buffer size or timeout values. However, fault avoidance falls short of the requirements for a truly dynamic responsive system. Value-domain errors are handled by standard voting protocols, where replica determinism is a necessary simplifying assumption.

The implementation of Delta-4 is based on local-area networks with hosts attached via specific, self-checking network attachment controllers. The network uses a custom-designed atomic broadcast protocol that allows the voting mechanisms to be easily implemented.

Delta-4 also includes an extensible management architecture that configures the redundancy level, monitors the system behavior, and takes care of fault treatment (including diagnosis and reconfiguration). Fault treatment works closely together with the error-processing protocols and is automated as far as possible.

Delta-4 does deliberately not propose a programming model of its own, but tries to integrate with the reference model for open distributed processing. This generality is paid for with a certain aloofness and inconcreteness in the architecture that also makes it very difficult for Delta-4 to make any quantitative statements.

3.5.2 Multicomputer Architecture for Fault Tolerance—MAFT

The goal of the Multicomputer Architecture for Fault Tolerance (MAFT) [145] is to design a system that provides highly reliable and high-performance computing in a real-time control environment; an aerospace control application has been a driving example in MAFT's development. It is a hybrid model, where much of the fault-tolerance functionality is implemented in hardware, assisted by system software. MAFT origins in SIFT, but places more emphasis on performance issues. A MAFT system is composed of several nodes, connected by a broadcast network. A node consists of an application processor and an operation controller (somewhat reminiscent of Spring). The operation controller is responsible for communication and synchronization with peer processors, scheduling, voting, error detection, and configuration.

A basic principle in MAFT is that every data item exchanged between processors is voted upon to avoid even Byzantine faults. More specifically, a data message produced by the application processor is sent via the broadcast network to all other processors, accepted by the operation controller, and voted on. Voting requires data items to be tagged to identify them, and processors must also be synchronized. Since MAFT allows multi-version programming to avoid even design faults, different voting functions can be selected for each data item (e.g., to allow for different numeric precision of different implementations). An application processor accesses remote messages via the controller and does not have to process them itself.

As usual, tasks are a unit of work in MAFT. Tasks are periodic with variable, yet bounded execution times. Decisions of local schedulers are also voted upon. If a faulty processor is detected by voting (with a Byzantine agreement to ensure a correct result even in the presence of a fault), it is excluded from the set of operating processors. Since all local schedulers have global knowledge, a reconfiguration can take place that assigns the tasks of an excluded processor to other, functional ones. An excluded processor remains active (although its results are ignored), and it can even be reintegrated if it has not produced false results for a certain time.

The simple task model in combination with graceful degradation of the schedulers allows MAFT to be used in real-time environments: if the schedulers detect that, after reconfiguration, some deadlines of tasks are no longer met, they can dynamically shed load by replacing some non-critical tasks with faster counterparts. However, the need for frequent voting actions makes special hardware necessary.

3.5.3 Mars and TTP

The Maintainable real-time system (Mars) [150] follows a very strict approach to integrate real-time and fault-tolerance requirements. Its basic tenet is that predictable performance under any anticipated circumstances, in particular peak load and faults, is of utmost importance to a real-time system. To be able to guarantee predictable behavior of the entire system, the behavior of all its components, including the applications, must be known in advance.

To be able to give this guarantee, a system design is used where on the one hand, the load of the system is carefully analyzed off-line: all applications with real-time requirements must have clearly bounded CPU and communication needs. On the other hand, the system design is such that for the Mars operating system, faults usually do not become visible, but are completely masked by the underlying hardware—as long as they conform to prespecified fault models.

This fault masking is realized by implementing the hardware as a set of interconnected components with fail-silent semantics. Fail-silence is achieved by employing active replication in each component, e.g., by duplicating a processing element and using self-checking circuitry. Hence, such a component either works correctly or does not interact with its environment at all. This is complemented by the use of redundant Mars buses to interconnect these components.

On top of such hardware, predictability is then implemented by planning all system activities in an off-line computed schedule. This schedule includes fixed times for operating system activities and application process activations. To ensure that this schedule can actually be executed, the only source of interrupts in a Mars system is the globally synchronized real-time clock. This time-triggered design additionally improves predictability in the presence of faults as it acts like a low-pass filter against events from the environment and therefore prevents the system from overloads caused by event storms. Similarly, access to the communication bus is completely preplanned and performed by a deterministic Time-Division Multiple Access (TDMA) regime. The communication subsystem of Mars has later been redesigned and extended as Time-Triggered Protocol (TTP) [151], following the same basic principles.

In return for a complicated static analysis of system, application, and fault scenarios, Mars thus achieves a splendid responsiveness: it indeed meets all deadlines with probability 1—but only if all (strict) assumptions are met. Some additional flexibility has later been added to Mars by means of mode changes, enabling Mars to switch between different states of operation [80].

3.5.4 Consensus for Responsiveness—CORE

Compared to Mars, the Consensus for Responsiveness (CORE) system [192, 228, 302] takes a very different approach to responsiveness. The fundamental observation of CORE is the fact that load and faults are independent parameters—while it is true that under faults, systems often generate additional load for fault handling activities, there is no inherent reason to do so. Based on this observation, CORE attempts to maximize the user-perceived responsiveness of a given service since, as has similarly been argued in Section 2.3, it is irrelevant to the user if a service fails because the system was in overload or because faults occured (which might, in turn, have caused the overload). This dissertation also shares the tenet of CORE that it is impossible to firmly guarantee anything under realistic fault assumptions, and that it is only possible to maximize probabilities.

Upon this basis, CORE develops a novel formulation of adaptive fault tolerance: As long as the system is only lightly loaded, it is acceptable to use expensive fault-tolerance mechanisms that can handle even complex, yet unlikely fault models such as Byzantine behavior. If the load on the system increases, it might be better to dedicate the system's limited resources to processing the tasks using a simpler, but more efficient fault model (e.g., a simple voting protocol)—better means here resulting in a higher user-perceived responsiveness. The superiority of this approach is demonstrated with some design examples [302].

To implement this adaptive fault tolerance, parameterized consensus protocols are used. Consensus is a well established mechanisms for fault tolerance [28, 190]. In CORE, the protocols are parameterized to be able to adapt their level of complexity to the current load situation.

Two applications serve as demonstrations for CORE: one replays music on a set of distributed machines, where the instruments assigned to a machine are transparently and in real time taken over by other machines if it is switched off [305]. The second one controls small robots that keep a seesaw in balance, even if weights are placed on the seesaw or some of the robots are switched off [304].

While sacrificing the illusive ideal of firm guarantees, CORE achieves large flexibility to adapt to changing environment and system parameters under realistic assumptions.

3.6 Focus on Quality of Service

The notion of Quality of Service has already been discussed in Section 2.2. Here, a more detailed overview of projects and systems that have been proposed to achieve Quality of Service in a distributed environment is given. As is the case with performance, network-centric issues (see Section 3.6.1 for LANs and Section 3.6.2 for supercomputer and SANs) and system-centric issues (Section 3.6.3) can be distinguished.

3.6.1 Network Quality of Service

A dichotomy exists between circuit-switched and packet-switched networks. It is comparatively easy to provide real-time delivery guarantees, a typical Quality-of-Service requirement, in a circuit-switched network: a fixed portion of network resources is set aside. This is not a satisfying solution for a network that is expected to carry a wide range of traffic with varying, often bursty characteristics [12]. Moreover, networks in both computer science and industry are typically packet switched in one form or another. Therefore, solutions are needed that allow packet-switched multi-hop networks to carry traffic with Quality-of-Service requirements (overviews can be found in [12, 155, 282]).

Such Quality-of-Service requirements can be manifold: bounded delivery time; bounded jitter, where jitter is the difference between longest and shortest packet delay; or bounded probability of packet loss are examples. Further Quality-of-Service metrics, which are also important for non-real-time applications, are average throughput and average packet delay.

Much research has been done on providing these Quality-of-Service guarantees at the network level. Typically, the metrics are defined as end-to-end characteristics, with two hosts running the respective parts of an application as the two ends.[6] Connection-oriented methods are commonly used to provide Quality-of-Service guarantees; connections are sometimes called streams [12, 42] or real-time channels [126] (with some slight nuances in definition), but the underlying concept is the same.

The three phases of connection setup, usage, and breakdown are involved in Quality-of-Service guarantees. During call setup, it must be decided whether the call can be admitted on the basis of the existing reservations and the call's own requirements. The admission test also necessitates routing this call through the network and making reservations along the way (such as processing capacity or buffer space in routers). Additionally, the impact of admitting this call on other calls has to be taken into account [155]. To actually decide whether or not to admit a call, the traffic models are quite important. In much of the theoretical foundations for call admission, Markov models are the basis. There is considerable amount of evidence, however, that such models are not a good description of actual traffic patterns [172, 222]. Additional research appears necessary here.

An established connection represents a promise by the network to the application to deliver the requested service, provided the application conforms to its traffic description. To ensure such a traffic behavior, the network can install traffic monitoring and shaping devices at its edge—otherwise a malicious or malfunctioning application can endanger guarantees given to other participants. A simple, well known example for such a shaping algorithm is the leaky bucket [284], and more sophisticated monitoring allows (in principle) adaptive behavior of both network and application. After a packet has been admitted to the network, deciding which packet to forward over a given link, in presence of different Quality-of-Service classes, is the link scheduling problem. A good overview of the work in this area can be found in [282]. It can also be argued that the routing algorithms used for Quality-of-Service guarantees bear more resemblance to circuit switching than packet switching [12].

Fulfilling Quality-of-Service requirements is further complicated by the possibility of failures in the network. Bounded delivery time for packets, a typical Quality of Service requirement, is therefore difficult to meet. It has to be replaced by a probabilistic upper limit on the packet delay. In multiple-access networks, e.g., like the broadcast medium used by TTP [151], node redundancy and statically allocated broadcast messages via TDMA can be used to mask faults. This is not possible in multi-hop real-time networks. One possible solution is to use forward-recovery techniques to send multiple copies of data along disjoint paths. The number of paths can be adjusted to select the desired success probability. Another possibility is to use the additional paths only as a cold standby, and upon detection of failure, resend data along such a backup path. These paths have to be reserved in advance to make sure all necessary resources are available. One exemplary solution for this can be found in [100], which also discusses some other approaches; SHA et al. [254] investigate the problem in the context of IEEE 802.6 metropolitan area networks.

[6] Or multiple hosts, in the case of multicast traffic.

3.6.2 Quality of Service of parallel computer networks

While a tremendous amount of work is available on routing in fixed-connection and multistage networks in parallel machines [93], only few results beyond the obvious are available for upper bounds on communication times in a possibly faulty network. Even if no faults are assumed, upper bounds are only rarely available [93, p. 81].

Some of the problems for such bounds are enforcing limits on packet injection or accounting for contention in the network (particularly in wormhole-routed networks). One possible router architecture to handle this last problem has been introduced by REXFORD et al. [238], who also give a good overview over other projects. However, apparently none of these projects is capable of handling failed routers or nodes and still guarantee packet delivery times.

A Quality-of-Service extension to Fast Messages, FM-QoS, is a mechanism to provide guaranteed latencies and bandwidth on top of the popular Myrinet SAN [59]. Leveraging Myrinet's programmable network interface, FM-QoS uses network feedback (in the form of "backpressure") to implement self-synchronizing communication schedules. This feedback achieves synchrony in the network interfaces of the nodes and thus avoids collisions in the switch, resulting in predictable latencies of at most 23 μs for a single-switch network.

In an experimental study, HILL et al. [104] use the Bulk Synchronous Parallel (BSP) programming model [287] to estimate g, the time it takes to communicate a single word between two processors, normalized with respect to processor speed. Results show that, e.g., the Cray T3D/T3E machines have a fairly uniform behavior with small standard variation and few outliers, which can be fitted well by a normal distribution. They also show that medium-sized configurations of SGI Origin or IBM SP-2 machines (about 6-8 processors) have considerably more outliers. Another interesting observation is that a network of workstations with non-switched Ethernet can improve its standard deviation of g by a random backoff approach for messages, but this concept results in a higher mean value of g [68].

3.6.3 Endsystem Quality of Service

Even a hypothetical interconnection network that gives perfect Quality-of-Service guarantees between two hosts is not sufficient to ensure that messages between distributed parts of an application are delivered on time. Within a time-shared endsystem, an application process usually has to compete for resources that are necessary to process, send, and receive messages. Examples for these resources are the Central Processing Unity (CPU), cache, memory and I/O bus, bridges between these buses, DMA engines, and the network interface itself. Both the total amount of these resources and their temporal availability are crucial, as is the enforcement of contracts or subscriptions by the operating system. It is therefore necessary to build systems that perform admission control to avoid over-subscription and that schedule the correct resources for the right message at the right time. To this end, a number of Quality-of-Service architectures have been proposed; overviews of these architectures can be found in [16, 208]. Some notable systems or proposals include the following.

- NAHRSTEDT and SMITH [210] clearly show the need for Quality-of-Service support in the end systems. They showed that operating system effects dominated any effects caused by the network in an experimental investigation of end-to-end Quality of Service. They observed that process priorities are of limited usefulness, must be coordinated among distributed applications, and integrated in the network protocol stack processing.

  Based on this work, the QoS Broker introduced by NAHRSTEDT and SMITH [211] focuses on multimedia applications (a telerobotics application was used as a test case) and coordinates the activities and resource requirements of multiple protocol processing layers. The broker negotiates with network management and remote brokers and selects runtime policies to implement an application's desired Quality of Service. For example, to bound jitter, the broker can decide whether to use more buffer space, tighten the jitter requirements on the network, or closely coordinate the execution of processes in time. Therefore, the broker has responsibilities of call admission and Quality-of-Service negotiation and translation.

This broker was later completed by the OMEGA endpoint architecture for provisioning of Quality-of-Service guarantees [209]. OMEGA essentially separates an application level protocol from a network level protocol stack as schedulable entities. From this architecture, some unexpected lessons were learned, e.g., the difficulties induced by a blocking DMA engine for transfers to the network interface, and that the "real-time" priorities of the underlying AIX UNIX are not sufficient for Quality-of-Service protocol processing.

- GOPALAKRISHNAN and PARULKAR [90] propose a Quality-of-Service framework for multimedia applications. It focuses on three main points: on Quality-of-Service specification at the application level, where it identifies an isochronous, a bulk data, and a low delay class; on Quality-of-Service mapping from application level to network level Quality-of-Service parameters; and on Quality-of-Service enforcement. A traditional mechanism for Quality-of-Service enforcement would be to assign threads to independently schedulable protocol operations and schedule them via real-time scheduling methods like EDF or RMS. However, doing so is inefficient since processing a single data unit can be less time consuming than the context switch to start a corresponding thread [56]. Instead of threads, this framework uses so-called real-time signals. These signals are scheduled using rate monotonic scheduling with delayed preemption [91], i.e., a handler invoked by a signal is preempted only at the end of an iteration, which processes a single data unit. An implementation mechanism for this framework is also discussed.

- MEHRA et al. [197] introduce a Quality-of-Service-sensitive communication subsystem architecture, which ensures (1) maintenance of Quality-of-Service guarantees, (2) overload protection, and (3) fairness to best-effort traffic. Real-time channels [126] are assumed as underlaying, Quality-of-Service-capable network. This subsystem is based on an *x*-kernel that has complete control of the CPU and can be implemented as a separate server running with suitable capacity reserves [169]. Such a server implementation, however, is somewhat in conflict with the need for user-level protocol processing necessary for high-performance protocols (see Section 3.2). Quality-of-Service contracts are enforced and messages are processed using a process-per-channel paradigm.[7] The host resources CPU bandwidth, link bandwidth, and buffer space are managed by such channel handler processes. For this architecture, MEHRA et al. [196] discuss tradeoffs between resource capacity and channel admissibility for real time, where preemption grain of CPU and links are significant parameters.

A lot of research has been performed, and much progress has been made. Two main themes can be observed: either an attempt to cope with given operating system environments and to manipulate them as far as possible (e.g., modifying priorities like in [209]), or modifying the operating system extensively. Most approaches of the latter kind share early demultiplexing of arriving packets and handling them by separately schedulable entities as a common technique—ideally supported by programmable network interfaces.

However, it is still too early to judge what are the best solutions for a given application. There is still a long way to go to developing a commonly applicable, generally available host architecture that is capable of providing an underlying network's Quality-of-Service guarantees to an application.

[7] As opposed to the process-per-protocol model sometimes used in UNIX or an also conceivable process-per-message model.

A theory has only the alternative of being right or wrong. A model has a third possibility: it may be right, but irrelevant.

– Manfred Eigen

Chapter 4

Problems in Responsive Cluster Computing—The Calypso Case

In Chapter 3, two parts of the Milan project, Calypso and Charlotte, have been identified as good candidates to build a responsive system for high-performance computing based on COTS components. The basic techniques and the implementation of these systems are described here in greater detail (see Section 4.1), concentrating on the Calypso system. A number of factors limiting Calypso's responsiveness are identified in Section 4.2, which are then remedied in the following chapters. The discussion of Charlotte is postponed to Chapter 9, where issues of metacomputing in wide area networks are examined. A sample Calypso program and some experiments are discussed in Section 4.3 and Section 4.4, respectively.

4.1 An overview of Calypso

Calypso [23] is a software system for parallel programming in cluster environments. One of the main objectives of Calypso is to provide a simple programming environment: A programmer should not have to worry about the complexities of an actual execution environment, like the number of available machines, the relative speed of machines, or potentially failing machines. It is much easier to write programs for an abstract, simplified, perfect environment. Calypso provides such an abstraction—a Calypso program is written for an idealized, perfectly reliable PRAM with an infinite number of processors. It is the responsibility of the Calypso middleware to implement this abstraction on a set of real machines, hiding the aforementioned complexities.

The theoretical background for the Milan project (of which Calypso is but one system) is work on efficient asynchronous execution of large-grained parallel programs (cp., e.g., [15]). Consider a program $\mathcal{P}$ that is written for an idealized, synchronous PRAM machine, using a BSP-like style [287]. $\mathcal{P}$ consists of a number of parallel steps, where each step has a number of parallel routines. Assume further that in one parallel step, any shared variable is updated by at most one routine. It is a challenging problem to execute such a program on a realistic, asynchronous machine (where a processor can also become infinitely slow, i.e., fail). This problem is solved in [15] by compiling $\mathcal{P}$ into a semantically equivalent program $C(\mathcal{P})$ that can be efficiently executed on an asynchronous machine.

To write programs for such an idealized PRAM, Calypso extends the C programming language with only four keywords: `shared`, `parbegin`, `parend`, and `routine`. `shared` is used to declare data as accessible from concurrently executing parts of the code. The keywords `parbegin` and `parend` together demarcate a parallel step: code inside such a parallel step can be executed in parallel, code outside is run sequentially. Hence, a Calypso program is an alternation of sequential and parallel steps. Within a parallel step, `routine` denotes one or more units of concurrent execution. A `parend` constitutes a barrier synchronization for all routines within a parallel step—the step ends once all its routines have terminated. This concept of barrier

synchronization for parallel routines captures the essential point of VALIANT's BSP programming model [287]. An example for such a parallel step would look like this:

```
parbegin
  routine[int-expr] (int width, int id) { routine body 1 }
  routine[int-expr] (int width, int id) { routine body 2 }
  ...
  routine[int-expr] (int width, int id) { routine body n }
parend
```

The *routine body* of a `routine` statement is the sequential code executed as a parallel routine. The optional *int-expr* argument of `routine` is the number of instantiations of such a routine. For the formal parameter *width* the actual number of routines at invocation time is inserted; *id* is the unique number for each routine. One routine is therefore able to identify its own identity relative to its sibling routines.

Such a routine can have local variables, and it can access global variables as shared memory if they are annotated with `shared`. The memory consistency model for such a routine is also very simple: local variables are initially undefined, and for all routines within a parallel step, the shared variables retain the value they had at the beginning of a parallel step. Access to shared data follows the Concurrent Read Exclusive Write (CREW) policy: a data item can be read by any number of routines, but only written by at most one. Programs with Concurrent Read Concurrent Write (CRCW) behavior are also executed correctly if all write accesses to a variable write a unique value. Write updates to shared data occur atomically at the end of a parallel routine. As a consequence, all routines execute in isolation, and updates to shared memory are only visible after a parallel step has finished. This isolation allows the programmer to ignore the order of execution of routines when writing the program. In particular, a read of any shared variable always returns the value the variable had at the beginning of a parallel step unless the variable has been modified locally within the routine itself. While this simple consistency model does not allow some optimizations enabled by relaxed consistency models, it is commensurate with the argument brought forward by HILL [105]: with modern processor's speculative execution, the complexity of relaxed models outweighs their possible performance benefits.

How can this semantics be implemented on real, unreliable machines? A Calypso program executes in a master/worker fashion. The master process executes all sequential steps and manages the execution of the parallel steps. The routines of these parallel steps are executed by any number of worker processes, usually residing on remote machines. At the beginning of a parallel step, the master waits for workers requesting work. Upon such a request, the master assigns a routine to a worker, which will then execute the routine. During execution, the worker will (usually) access data in the shared memory. The shared memory is implemented at the page level: at the beginning of a parallel step, all the pages of the shared memory are protected against access. If a worker reads a variable located on such a page, the operating system raise a page fault exception. The Calypso library catches this exception and requests the corresponding page from the master. A write access to a protected page marks it as dirty with a similar mechanism, and at the end of a routine, all dirty pages are sent back to the master.

The master, however, cannot yet integrate such a dirty page in the shared memory, since then a request for this page by another worker would possibly result in a value different from the one at the beginning of a parallel step. Hence, the page updates from the workers can only be included in the master's shared memory when a parallel step has been completed. This memory management is called Two-phase Idempotent Execution Strategy (TIES).

TIES enables another important technique: eager scheduling. It is at the core of Calypso's mechanism for load balancing and tolerating worker failures. If a worker fails the routine it has been assigned will not be finished, and the master would wait for this routine, even though other workers become idle. In Calypso, the master can re-assign already started routines to idle workers (once all routines have been assigned at least once), since TIES guarantees an idempotent, exactly-once semantics of routine execution. This reassignment implies that worker failures can be masked, and that slow machines do not stall a computation, since eager scheduling entails automatic load balancing. Even intermittently available machines can be used by Calypso.

Calypso allows some additional performance optimizations (described in more detail in [22]) that take advantage of special semantics of a program. One such optimization is to allow updates of memory to be applied to the master's copy as soon as a routine finishes, and not only at the end of an entire parallel step. This optimization does not violate the TIES semantics if the read and write sets of data in all routines in that particular step are disjunct. The advantage of this immediate update is that it heavily reduces the master's state size during a parallel step; it makes the state even of constant size. Other such optimizations are handing out routines to workers in larger chunks than just single routines and trying to optimize routine placement on workers. A followup project to Calypso, Chime [245], allows, among other things, nested parallelism and the shared use of stack variables even by routines running on distributed machines [246].

4.2 Responsiveness shortcomings of Calypso

Calypso does provide mechanisms for tolerating crash faults of workers. But this is not enough to be able to make a convincing argument for Calypso as a responsive system. Design and implementation shortcomings that limit Calypso's responsiveness must be identified.

4.2.1 Need for an analysis of eager scheduling

In determining the responsiveness or, more generally, the response time distribution RTD of a Calypso program, it is necessary to know the RTD of its constituent parts. Estimating runtimes for sequential programs or bounding the runtime of an algorithm on a language level is a well researched area in the real-time community. Calypso is further complicated by the parallel step with its eager scheduling execution mechanism. The eager scheduling is at the core of Calypso's fault tolerance and must therefore be accurately analyzed with regard to its RTD.

Since estimation of runtimes (with or without faults) is in general an undecidable problem, proper assumptions have to be made. In Chapter 5, a refined model of a parallel step in Calypso is given and a mathematical analysis of its runtime distribution is performed.

4.2.2 Removing a single point of failure

An obvious shortcoming of Calypso's design is the master process, which constitutes a single point of failure and as such can be a serious obstacle to high responsiveness. There are a number of established fault-tolerance techniques to deal with such situations. Two popular ones, with quite different characteristics, are checkpointing and replication. The former can be regarded as an example for redundancy in time, the latter as one for redundancy in space.

Checkpointing has the advantage of being based on a simple, straightforward model. Accordingly, it has been investigated under numerous different perspectives. In the context of responsiveness, it is important to choose checkpointing parameters so as to maximize the responsiveness of a given program invocation. The only parameter that can be chosen (for a given program) is the interval of writing checkpoints. For this parameter, a general analysis that allows to maximize the responsiveness of a checkpointed service in a somewhat simplified mathematical model (e.g., the interactions with worker processes is not taken into account) is given in Chapter 6. To accurately judge the impact of checkpointing on a master/worker system like Calypso, an experimental investigation of this problem is also presented in Chapter 6.

Replication can not only serve as a fault-tolerance mechanism, it also enables the sharing of load between the replicated masters. Load sharing requires fast communication between workers and masters. Additionally, the semantics of a single program executing in a consistent fashion must not be violated. In particular, the interface of this program with its environment, namely its input/output behavior, must not change. Therefore, replicating a master really requires solving two different problems: One problem is the interaction of the replicated masters with each other, which is quite specific to Calypso. The other problem is providing a uniform, yet fault-tolerant solution to handling the I/O of a set of replicated processes. Such a general solution as well as a replicated Calypso master are discussed in Chapter 7.

4.2.3 Guaranteed resource allocation for parallel programs

One of the motivations behind the particular design of Calypso's fault-tolerance mechanism was its intended use in a cluster of workstations that is used both for parallel programs and in interactive use, making unused processing cycles of interactive workstations available to parallel programs. However, such workstations also have used cycles: interactive users generate an unknown, possibly stochastically describable load on their machines. Such interactive load is in stark conflict with goals of responsive execution of the parallel routines, since it is not clear how much resources are available for the parallel program at any given time—and with insufficient resources, it is impossible to meet deadlines (for a fixed program).

An alternative to a stochastic description of user behavior is a contract between interactive user and parallel program: a certain share of resources is reserved for the parallel program, and the user is also assured that this program will not exceed its share. A few systems implement such contracts; however, directly using these systems for parallel programs results in unacceptable performance losses. A solution how such contracts can be efficiently implemented for parallel programs is presented in Chapter 8.

4.2.4 Communication overhead and reaching remote resources

For any distributed program, the amount of communication can have a large impact on its responsiveness. Using information about the communication requirements of a Calypso program to reduce the communication has only little effect on program runtime (the administration costs outweigh the reduction in communication costs). For wide area networks, however, this tradeoff is different. With the increasing performance of wide area networks, metacomputing becomes an increasingly attractive prospect. The Charlotte system targets such metacomputing environments. It is shown in Chapter 9 how using information about the communication pattern of a Charlotte program can improve its performance significantly.

4.3 A simple Calypso program

As an aid to perform some experiments, the following simple model for a Calypso test program is used:

- The sequential steps are neglected, the program consists of a loop over a parallel step.
- The number of executions of this step is denoted by s.
- The number of routines in a single parallel step is fixed for all iterations and denoted by n.
- The average length of a single routine (the granularity) is indicated by g (in seconds).
- The imbalance between these routines is given by v, expressed in percent of granularity. The runtime of a single routine is then a random variable with uniform distribution $\mathcal{U}(g - (g * v)/2, g + (g * v)/2)$.
- The amount of traffic between master and worker, parameterized with a,[1] the number of pages (of 4 KBytes each) that each routine has to read and write, in addition to some small status messages that are also exchanged between master and worker.
- The number of workers m used to execute the program.

In most of the following experiments, the product ng is often fixed to better see the impact of varying granularity. Note, however, that this implies that the total amount of traffic between master and workers increases since it is (roughly) proportional to $n(a + 1)$.

Since Calypso implements a programming model similar to BSP, these parameters can also be considered as a description of a BSP program [287]. In Chapter 8, Calypso and BSP-style programs (as described by these parameters) are used for experiments.

[1] a for "amount". The more intuitive t for traffic is needed to represent time.

4.4 Some experiments

Based on this example program, a few experiments are presented in this section to give an impression of the performance of Calypso programs. These experiments were executed on four Pentium 90 machines, interconnected by a standard 10 Mbps Ethernet. The number of routines n was chosen so that $ng = 1$ s while g is varied between 1 ms and 100 ms. All numbers are averaged over 50 executions.

Figure 4.1 shows the impact of granularity on the execution time without additional traffic ($a = 0$), Figure 4.2 for $a = 1$, and Figure 4.3 for $a = 5$. Calypso behaves as expected: for low to moderate traffic, it scales as long as the granularity is not too small. The higher the granularity, the lower are the inherent overheads, but the lower are also the benefits of load balancing. For a single worker, the total execution time is longer than 1s owing to the additional traffic (depending on a) and the inherent overheads of Calypso (administrating the routine table, etc.).

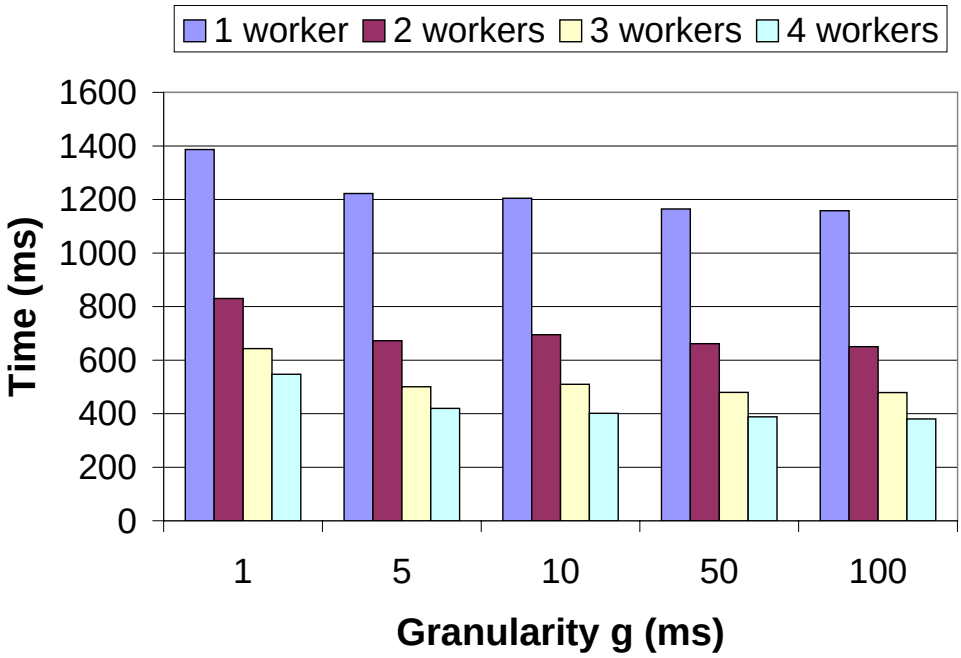

Figure 4.1: Average runtime of a single parallel step with varying granularity g and number of workers m, other parameters $a = 0$, $v = 0$, $ng = 1$ s.

Figure 4.4 shows the impact of imbalance in the routines on the total execution time. Even for severely imbalanced parallel steps, Calypso's load balancing mechanisms manage to hide that imbalance and achieve rather uniform total execution times. The effects of load balancing only start to degrade slightly at 100 ms granularity. The results are similar for other values of a and m. Hence, imbalance can be neglected as a parameter in practical experiments.

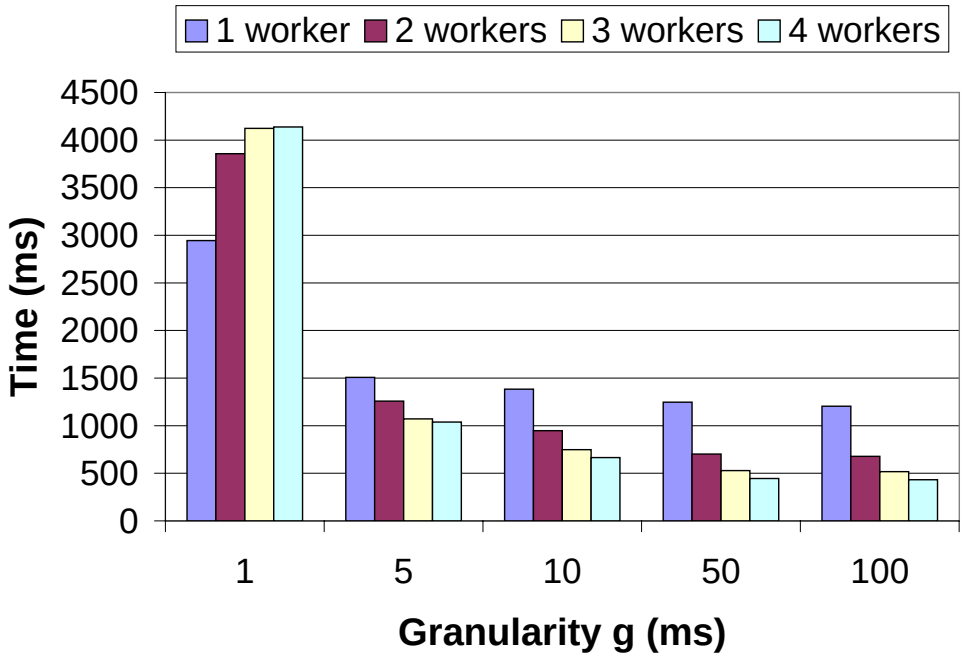

Figure 4.2: Average runtime of a single parallel step with varying granularity g and number of workers m, other parameters $a = 1$, $v = 0$, $ng = 1$ s.

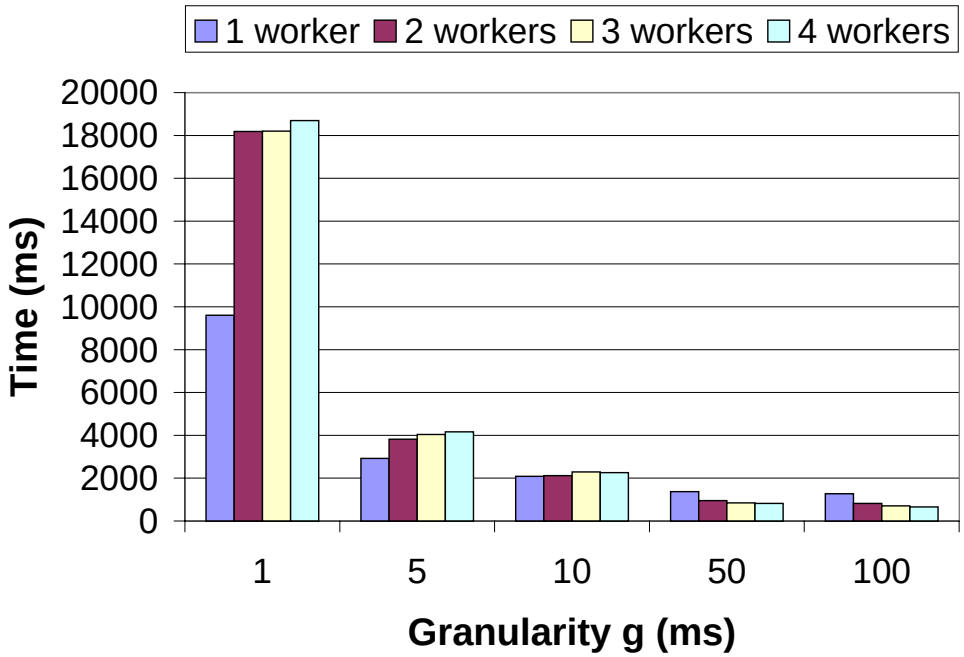

Figure 4.3: Average runtime of a single parallel step with varying granularity g and number of workers m, other parameters $a = 5$, $v = 0$, $ng = 1$ s.

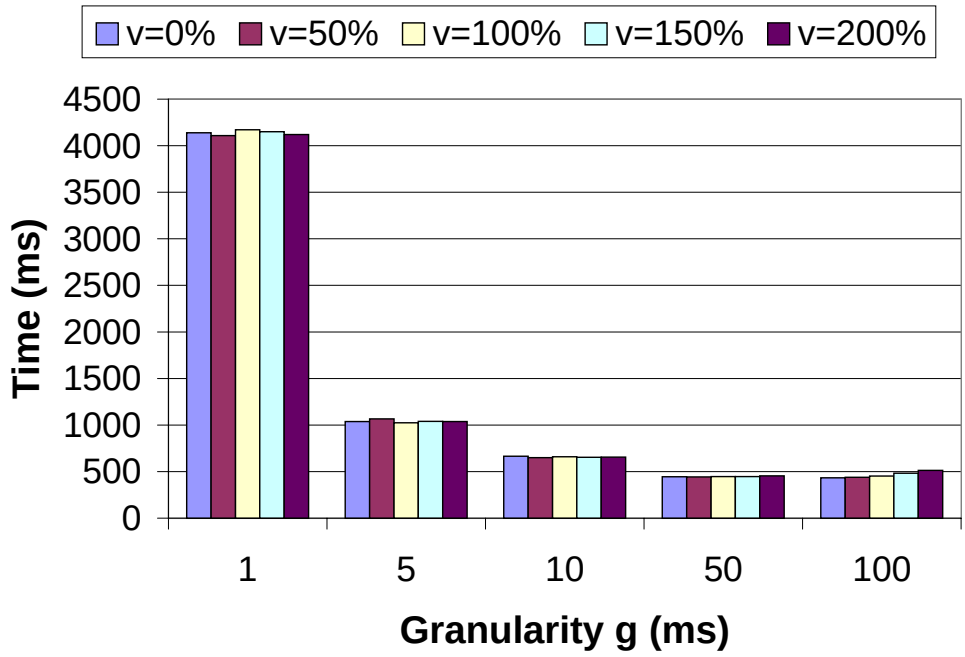

Figure 4.4: Average runtime of a single parallel step with varying granularity g and imbalance v (in percent), other parameters $a = 1$, $m = 4$ workers, $ng = 1$ s.

It is today very difficult to write mathematical books. If one is not concerned with the finer points of theorems, explanations, proofs, and conclusions, it will not be a mathematical book; but if one does, it will make for utterly boring reading. . . . And even the most detailed explanation has its darkness, none less than laconic brevity.

– Johannes Kepler, Astronomia Nova, 1609

Chapter 5

Analysis of Eager Scheduling

In this chapter, the runtime distribution of a parallel step with eager scheduling as execution regime is analyzed. Worker machines can be of different speeds and possibly experience faults; the routines of a parallel step are modeled with two different assumptions, leading to two differently complex ways of analyzing the problem.

5.1 Introduction

In order to reason about or even guarantee the responsiveness of a Calypso program, it is necessary to know the responsiveness of its parallel and sequential steps. For sequential steps, this is a well researched problem: estimating runtimes of programs (see for example [50, 234, 275]). A parallel step is a generalization of a single sequential program since it consists of many concurrently executing sequential program parts.

In general, computing the runtime distribution of even a sequential program is an undecidable problem, since it generalizes the halting problem. It is hence necessary to limit the scope of this problem to make it tractable. The choice of assumptions about the program behavior results in different levels of complexity in the analysis, and also determines the level of realism of a model. Given sufficiently realistic assumptions, such an analysis allows probabilistic guarantees about the execution times of a parallel step and, ultimately, an entire Calypso program. In this chapter, an analysis of the response time distribution of a parallel step when eager scheduling is used is presented; two different assumptions about the behavior of individual routines are considered.

In an environment with fixed resources (e.g., only one available processor) and a fixed program that has a deadline, there is little that can be done beyond executing the given program and hoping that it will finish before its deadline. It is, however, quite evident that the amount of available resources has a considerable impact on the runtime distribution of a program. If it is possible to compute the runtime distribution as a function of the available resources, and if the execution environment of the parallel program is capable of adapting to a varying number of resources, a resource management system can use this information to make informed decisions how much resources to dedicate to a given program. Work on resource management systems abounds, a candidate for cooperation with a Calypso program is the Just-in-Time resource manager described in [24].

Conversely, a program that is capable to tune its resource requirements to the given resource situation can meet deadlines, possibly by sacrificing quality or precision, where a fixed program would have failed—

CHANG et al. [49] describes such a tunability extension for Calypso. However, for a program to correctly judge the results of tuning its execution, an analysis of the effects of the alternatives with regard to their runtime distribution is again needed. This necessitates a fast and simple computation of the runtime distribution; a goal which is in conflict with an also desirable generality of assumptions. This problem of generality is discussed in more detail in Section 5.4.

In the remainder of this chapter, an overview of related work is given in Section 5.2. Suitable assumptions about the runtime behavior of parallel routines in a detailed model of eager scheduling are discussed in Section 5.3, and the analysis of eager scheduling's runtime distribution is presented in Section 5.4. In Section 5.5, a few examples of runtime distributions as derived by this analysis are shown. Finally, some conclusions are presented in Section 5.6 and possible extensions are considered in Section 5.7.

5.2 Related work

The problem of scheduling parallel jobs has been studied extensively. Mostly, mean execution time or simple bounds on minimum or maximum execution time have been given. Also, the problem of analyzing eager scheduling has not appeared in the literature so far; usually much simpler scheduling strategies were under scrutiny. An overview of results that emphasizes performance-oriented scheduling can be found in [74].

In a classic paper, ULLMAN [285] investigates the problem of scheduling tasks on processors under a complexity-theoretic point of view. Extending on earlier result that shows that already the scheduling problem for tasks with arbitrary lengths on two processors is NP-complete (when the total execution time is to be minimized), ULLMAN proves the following two problems to be NP-complete as well: The first problem is scheduling tasks with unit length on an arbitrary number of problems,[1] the second problem is scheduling tasks with either one or two time units execution time on two processors. In particular this second problem is closely related to the problem analyzed in Section 5.4.4 (where faults are additionally considered) and shows that it is unrealistic to hope for an efficient, optimal algorithm.

YAZICI-PEKERGIN and VINCENT investigate in [308] the execution of a task graph on an infinite number of processors. Nodes in the graph represent tasks with independent and identically distributed execution times; various distributions are considered. No contention for shared resources is assumed, therefore only the impact of synchronization is evaluated. Lower and upper bounds on total mean execution time are derived, depending on the topologically critical path (lower bound) and on a deliberately incorrectly assumed independence of paths (upper bound). However, the infinite number of processors makes this analysis less attractive for practical settings.

In [187], MADALA and SINCLAIR consider divide-and-conquer and multiphase (sequential step/parallel step type) algorithms, the latter is equivalent to Calypso's model. Execution times for the individual tasks are random variables with known distributions, mean execution time is evaluated and several types of upper and lower bounds are proven with an (effectively) unlimited number of processors. In addition, the average execution time under static and dynamic scheduling (with more tasks than processors) is bounded and the optimum number of parallel tasks to be created to minimize the execution time bounds (not the execution time itself) with constant scheduling overhead is determined.

JAIN and RAJARAMAN [117] give lower and upper bounds for the minimal time to execute a Directed Acyclic Graph (DAG) of tasks for a given number of processors. This paper also proves lower/upper bounds on the minimum number of processors needed to process the DAG in the minimum possible time. The main contribution of this paper is a method to partition a task graph to make it tractable. Therefore, it does not really pertain to the eager scheduling problem, since eager scheduling's task graph is very simple.

LI [176] is concerned with DAGs where the nodes represent tasks and task execution times are independent, identically distributed random variables. Stochastic upper and lower bounds for mean execution time with a limited number of processors and arbitrary task graphs are presented and special problems for multiphase graphs, partitioning algorithms and linear pipelines are considered in more detail. Scheduling and communication overheads are ignored. The idea of this paper is to cut a complicated task graph into slices

[1] Note that this problem is solvable in polynomial deterministic time for two processors.

(corresponding to synchronization layers), get the (trivial) bounds on the slices and combine them. Also, bounds are tightened with information on the probabilistic density function of task execution times.

LI and SUN [178] discuss how to scale a parallel problem size when the number of processors in a machine is increased to obtain a constant average speed (average work per processor). The problem is represented by a task graph and task execution times are described by independent, identically distributed exponential random variables.

A gracefully degrading fault-tolerance mechanism for the Bulk Synchronous Parallel (BSP) model [287] is introduced in [247]. On a massively parallel machine, the BSP model is implemented with randomized, duplicated shared memory. Failures of nodes are detected at synchronization time, and a reconfiguration phase is initiated. The model assumes in particular that no nodes fail during reconfiguration and that node failures do not change the network performance. Under these assumptions expressions for performance degradation and overhead are derived. For most cases, the analytical results match experiments reasonably well. Under certain circumstances, however, the hash functions that implement the duplicated memory break down and do not generate a uniform distribution of data in memory, resulting in much higher runtimes in the experiments than predicted analytically. This model also suffers, compared to eager scheduling, from the need to explicitly detect failures and from a loss of valid work upon failure detection (the reconfiguration even discards results of processors that have not failed). However, it does not have a single point of failure like Calypso. Another simulation technique for BSP programs on machines where a constant fraction of faulty processors is acceptable is described in [147].

On a more technical level, MUPPALA and TRIVEDI [207] extend the performability notion of MEYER [201] and consider an on-line transaction processing problem with hard and soft deadlines. This problem is modeled using an extension of stochastic Petri nets, stochastic reward nets. This reward net is transformed into a Markovian model whose response time distribution is analyzed numerically; no closed-form solutions are given. The technique is intriguing yet rather complicated.

SIEGEL [258] investigates bounds for eager scheduling in a competitive analysis with an emphasis on easy computability, but does not consider faults. This chapter contains a complete analysis of the runtime distribution of a parallel step using eager scheduling in the presence of faults.

5.3 Model definition

For a probabilistic analysis of eager scheduling, the model of Section 4.3 has to be slightly extended to allow for more general assumptions. In particular, instead of only considering the mean execution time and imbalance of routines, the analysis considers an arbitrary random distribution of execution times. Since the execution time depends on the actual speed of the machine used to execute the routine, this execution time is given with respect to a machine of arbitrary speed 1. In a sense, it is comparable to the number of instructions of routine i, and is represented by the random variable (rv) I_i (I for instruction) for each of the $i = 1, \ldots, n$ routines. The I_i's are assumed to be non-negative, independent, but not necessarily identical; f_{I_i} is the density function for rv I_i. For technical reasons, the f_{I_i}'s are assumed to be continuous—discrete densities would demand some changes in the notation of the proof.

Since one of the main reasons for undertaking a probabilistic analysis is the possibility of failure, the dependability of machine j must also be considered. More concretely, the analysis will use the reliability function of each machine: assuming that a machine works correctly at the beginning of a parallel step, what is the probability that it still works after t time units? This is represented by the random variable S_j (S for survival), such that $\Pr(S_j > t) = 1 - F_{S_j}(t)$ is the reliability of machine j and F_{S_j} is its lifetime distribution [283, p. 118]. These S_j are explicitly assumed to be independent, but not necessarily identical.

The relative speed of machine j is expressed by c_j for each of the m machines. The master process of Calypso is assumed to execute on an additional, infinitely fast machine. Initially, the master's machine is assumed to be perfectly reliable; this restriction is removed in Section 5.4.5.

The principle of eager scheduling allows a number of different varieties. Here a deterministic one is considered (called "ordered eager scheduling") in which the array of unassigned routines is always searched

in increasing numerical order, and already assigned routines are searched in decreasing order.

The analysis also requires a number of simplifying assumptions. In particular, neither the communication between master and workers nor contention for any resources (e.g., queuing delays in the master when workers ask for data) is considered. Machines are dedicated to the parallel program with unchanging effective speed (the c_j are constant); Chapter 8 shows how such a virtual machine environment can be implemented on a real, shared cluster. Also, machines are only allowed to crash (a fail-silent fault model), and are not repaired after the crash. The communication overhead can, to a certain degree, be assumed to be expressed by I_i.

A summary of parameters used in the following analysis is given in Table 5.1

parameter	index range	description
n	N/A	number of routines in one step
I_i	$i \in \{1, \ldots, n\}$	independent rv, number of instructions of routine i
m	N/A	number of worker machines
c_j	$j \in \{1, \ldots, m\}$	relative speeds of worker machines
S_j	$j \in \{1, \ldots, m\}$	independent rv of machine j's lifetime

Table 5.1: Summary of input parameters for analysis of eager scheduling.

5.4 Analysis

In this Section 5.4, the actual analysis of the runtime distribution of a parallel step under eager scheduling is performed. To simplify the exposition, the special case of three routines executed on two machines is discussed in Section 5.4.1, illustrating the technique. In Section 5.4.2, this example is generalized to a solution for an arbitrary number of routines executing on two fault-free machines, and in Section 5.4.3, the worker machines can also potentially fail. In Section 5.4.4 it is shown how the efficiency of computing the analytical results can be significantly improved and complexity reduced if stronger assumptions are made, additionally allowing an arbitrary number of machines to be considered. And finally, the assumption that the master machine has to be perfectly reliable is dropped in Section 5.4.5.

For ease of notation and for a simpler exposition, it will be assumed here without loss of generality that $c_1 = 1$ and $c_2 \geq c_1$. Note that this assumption is not an inherent limitation of the analysis and could be easily removed.

5.4.1 A simple special case

Consider the case of $n = 3$ routines, executing on $m = 2$ worker machines. We will attempt to compute $\Pr(Z \leq t)$, where Z is a random variable denoting the time of successful completion of a parallel step under eager scheduling.

The eager scheduling algorithm starts by placing Routine 1, which has runtime x_1 with probability $f_{I_1}(x_1)$, on Machine 1. This routine will finish, assuming Machine 1 does not fail first, at some unknown time $x_1/c_1 = x_1$. The scheduler will also assign Routine 2 (having runtime x_2 with probability $f_{I_2}(x_2)$) to Machine 2, where it will finish at time x_2/c_2 (again assuming that no fault occurs on Machine 2).

Now two cases must be distinguished: $x_1 < x_2/c_2$ or $x_1 > x_2/c_2$ (an overview over the following case distinction is shown in Figure 5.1).[2]

1. $x_1 < x_2/c_2$

2. $x_1 > x_2/c_2$

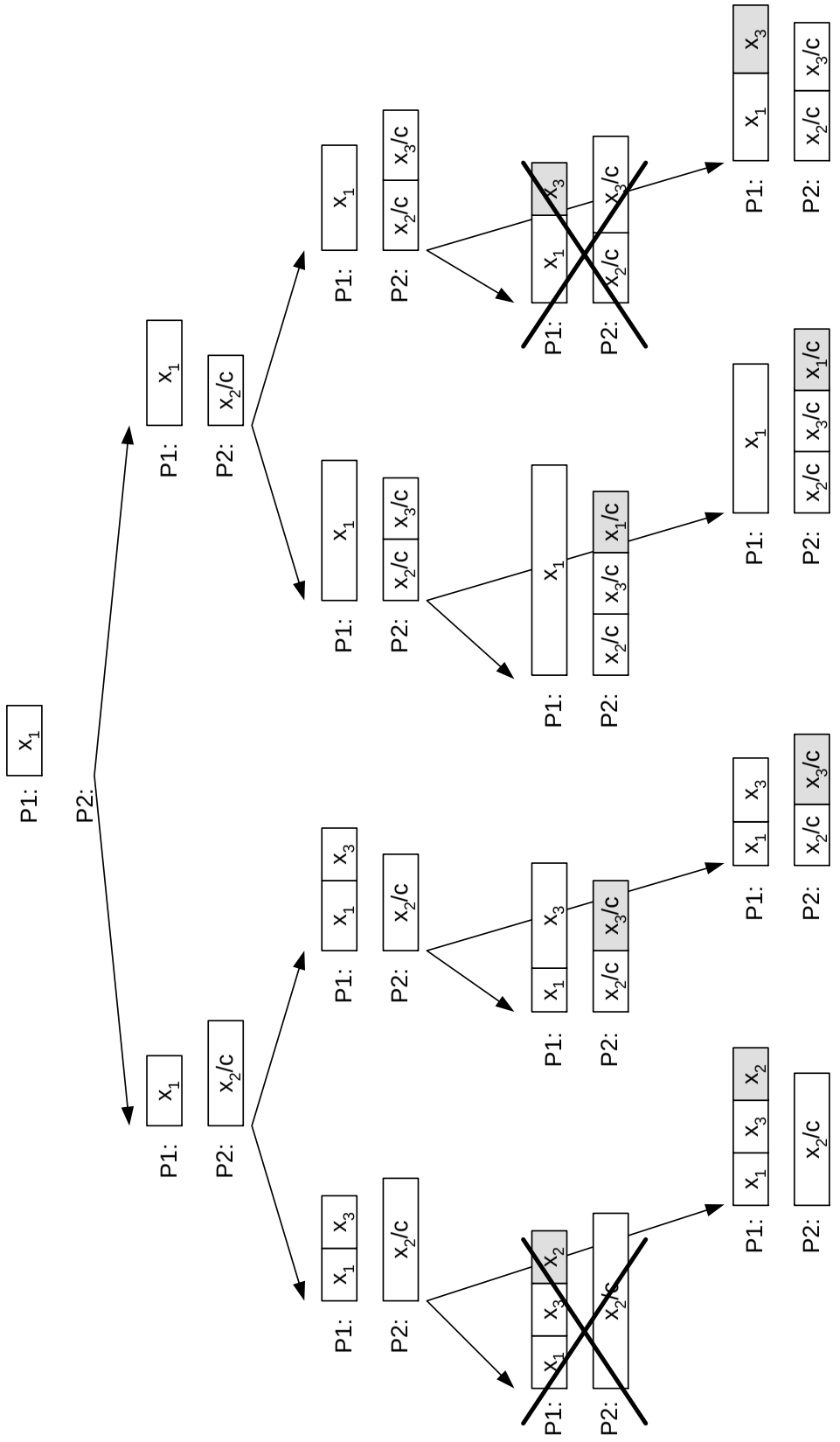

Figure 5.1: Overview over possible cases for eager scheduling of three routines on two machines (P1, P2). Arrows indicate scheduling steps, grayed boxes eagerly scheduled routines, and crossed out cases do not appear for $c_2 > c_1$.

After the first routine has finished, the scheduler will assign Routine 3 to the first machine asking for work. Routine 3 has runtime x_3 with probability $f_{I_3}(x_3)$. And again, the machine that has been assigned Routine 3 will become idle either before or after the other machine. Hence there are four cases so far:

1. $x_1 < x_2/c_2$ and $x_1 + x_3 < x_2/c_2$,
2. $x_1 < x_2/c_2$ and $x_1 + x_3 > x_2/c_2$,
3. $x_1 > x_2/c_2$ and $x_1 > x_2/c_2 + x_3/c_2$,
4. $x_1 > x_2/c_2$ and $x_1 < x_2/c_2 + x_3/c_2$.

Note how adding one routine generates an additional inequality that bounds the runtime for the new routine by a linear combination of the runtimes of the previously scheduled routines. The bound is either from above or below, and the corresponding lower or upper bound is either 0, or $\max\{c_1, c_2\}t = c_2 t$.

Now the "eager" part of eager scheduling comes into play. The first idle machine will be assigned a non-completed routine, which can be any of the three, but which is uniquely determined by the relative lengths of the three routines—as long as there are no faults. This extends the previous cases as follows:

1. $x_1 < x_2/c_2$ and $x_1 + x_3 < x_2/c_2$, Routine 2 is eagerly scheduled on Machine 1,
2. $x_1 < x_2/c_2$ and $x_1 + x_3 > x_2/c_2$, Routine 3 is eagerly scheduled on Machine 2,
3. $x_1 > x_2/c_2$ and $x_1 > x_2/c_2 + x_3/c_2$, Routine 1 is eagerly scheduled on Machine 2,
4. $x_1 > x_2/c_2$ and $x_1 < x_2/c_2 + x_3/c_2$, Routine 3 is eagerly scheduled on Machine 1.

It depends on the actual length of the routines and on c_2 whether or not the eagerly scheduled routine terminates before or after its first instance; both cases are possible. Hence we now have eight cases:

1. $x_1 < x_2/c_2$ and $x_1 + x_3 < x_2/c_2$, Routine 2 is eagerly scheduled on Machine 1, and $x_1 + x_3 + x_2 < x_2/c_2$ (this is impossible for $c_2 > c_1$),
2. $x_1 < x_2/c_2$ and $x_1 + x_3 < x_2/c_2$, Routine 2 is eagerly scheduled on Machine 1, and $x_1 + x_3 + x_2 > x_2/c_2$,
3. $x_1 < x_2/c_2$ and $x_1 + x_3 > x_2/c_2$, Routine 3 is eagerly scheduled on Machine 2, and $x_1 + x_3 > x_2/c_2 + x_3/c_2$,
4. $x_1 < x_2/c_2$ and $x_1 + x_3 > x_2/c_2$, Routine 3 is eagerly scheduled on Machine 2, and $x_1 + x_3 < x_2/c_2 + x_3/c_2$,
5. $x_1 > x_2/c_2$ and $x_1 > x_2/c_2 + x_3/c_2$, Routine 1 is eagerly scheduled on Machine 2, and $x_1 > x_2/c_2 + x_3/c_2 + x_1/c_2$,
6. $x_1 > x_2/c_2$ and $x_1 > x_2/c_2 + x_3/c_2$, Routine 1 is eagerly scheduled on Machine 2, and $x_1 < x_2/c_2 + x_3/c_2 + x_1/c_2$,
7. $x_1 > x_2/c_2$ and $x_1 < x_2/c_2 + x_3/c_2$, Routine 3 is eagerly scheduled on Machine 1, and $x_1 + x_3 < x_2/c_2 + x_3/c_2$ (also impossible for $c_2 > c_1$, similar to Case 1),
8. $x_1 > x_2/c_2$ and $x_1 < x_2/c_2 + x_3/c_2$, Routine 3 is eagerly scheduled on Machine 1, and $x_1 + x_3 > x_2/c_2 + x_3/c_2$.

[2]The case $x_1 = x_2/c_2$ has probability 0 for any distributions with continuous densities, which we have assumed above. In a discrete density case, ties can be broken arbitrarily.

In case a fault occurs, the other machine has to complete all unfinished routines. So conceptually, the above schedules can be extended by appending all routines that have not been scheduled on a given machine to this machine. The schedule finishes when all routines have been completed. Since a processor can (potentially) be assigned all three routines, it can fail during the execution of the first, second, or third routine or only fail after all three routines have been completed—this number of routines that a given machine j survives is indicated by s_j. Of course, at least one processor must survive until the schedule is completed. Hence, for all the eight cases shown above, there are a number of subcases that enumerate the possible fault combinations and determine their respective termination time. Also note that after the first eager scheduling step has occurred, the relative execution times of tasks on the two machines is of no consequence since these redundant assignments are only executed if one machine has failed.

As an example, consider the last case from the eight cases shown above. Conceptually, Routine 2 is additionally scheduled on Machine 1, and Routine 1 is added to Machine 2, to compensate for a potential failing of the other machine already during its very first routine. Table 5.2 gives an overview of the termination times for this case with all pertaining combinations of faults. For the other cases, the termination times in this table would be different.

	0	1	2	3
0	fails	fails	fails	$x_1 + x_3 + x_2$
1	fails	fails	$x_1 + x_3$	$x_1 + x_3$
2	fails	$x_2/c_2 + x_3/c_2$	$x_2/c_2 + x_3/c_2$	$x_2/c_2 + x_3/c_2$
3	$x_2/c_2 + x_3/c_2 + x_1/c_2$	$x_2/c_2 + x_3/c_2$	$x_2/c_2 + x_3/c_2$	$x_2/c_2 + x_3/c_2$

Table 5.2: Successful termination times of the various subcases of Case 7. Columns indicate the number s_1 of routines that Machine 1 survives, rows indicate s_2.

Let us now begin to compute $\Pr(Z \leq t)$. We are going to look at all possible combinations of x_1, x_2, and x_3. For such a combination, we compute the probability of its occurrence. Given such a combination, we consider the combination of fault scenarios (s_1, s_2) that can occur when the routines are executed. Hence, the law of total probability lets us start with the formulation ($\vec{x} = (x_1, x_2, x_3)$ and $\vec{s} = (s_1, s_2)$):

$$\Pr(Z \leq t) = \int_{x_1,x_2,x_3 \in (0,\infty)} f_{I_1}(x_1) f_{I_2}(x_2) f_{I_3}(x_3) \sum_{\vec{s} \in S} \Pr(\vec{x}, \vec{s}) h(t, \vec{x}, \vec{s}) \mathrm{d}x_3 \mathrm{d}x_2 \mathrm{d}x_1,$$

where S is the set of fault combinations, $\Pr(\vec{x}, \vec{s})$ is the probability that a certain fault scenario occurs for a given combination of x_i, and h is a function that tests if the execution of the three routines with the given runtimes succeeds before time t under a given fault scenario $\vec{s}$. Of course, c_2 is an implicit parameter of h.

Since the function h basically requires the implementation of an eager scheduling algorithm, we want to break this down into simpler functions. To do so, we take advantage of the case distinctions introduced above.

The important point to note here is that any of the eight cases as defined above determine a subset of routine combinations such that all combinations in this set have the same behavior; namely, their scheduling order on the two given machines is the same. Moreover, as we have noted above, each additional routine introduces one additional inequality that can be used to bound its value, and can be directly used as a limit for the corresponding integral. The other bound is either 0 or infinity, where we will refine infinity as an upper bound later.

The situation is complicated by the inequality introduced by the first eager scheduling step. This inequality cannot be directly mapped onto an integral. However, we can express this inequality by means of the Heaviside function: $a < b \Leftrightarrow H(b - a) = 1$, where $H(x) = 1 \Leftrightarrow x \geq 0$ and 0 otherwise.

This allows us to refine the above expression for $\Pr(Z \leq t)$ as follows (for Case 8 as an example):

$$\Pr(Z_8 \leq t) = \int_{x_1=0}^{\infty} f_{I_1}(x_1) \int_{x_2=0}^{c_2 x_1} f_{I_2}(x_2) \int_{x_3=c_2 x_1 - x_2}^{\infty} f_{I_3}(x_3) \\ H((x_1 + x_2) - \frac{x_2 + x_3}{c_2}) \sum_{\vec{s} \in S} \Pr(\vec{x}, \vec{s}) h(t, \vec{x}, \vec{s}) \mathrm{d}x_3 \mathrm{d}x_2 \mathrm{d}x_1$$

and similarly for the other seven subcases. As can be seen from Table 5.2, the set of fault scenarios S is just the set $\{(0,3),(1,2),(1,3),(2,1),(2,2),(3,0),(3,1),(3,2),(3,3)\}$; all other fault scenarios do not result in a successful completion of the parallel step.

Instead of using ∞ as upper limit, $c_2 t$ is also sufficient, since no routine longer than this has any chance of being completed before t, even if it runs alone on the faster of the two workers. The feasibility test function h only means comparing the termination time for this case and fault scenario to the actual time t. At this point, it is straightforward to write down the complete expression for the runtime distribution for this example. For a complete, rather lengthy, expression, please refer to [134].

5.4.2 General solution for two fault-free machines

To generalize the solution from Section 5.4.1 to an arbitrary number of routines, we are faced with the following

Problem 5.1 (Runtime distribution for eager scheduling). *Given n routines with continuous density functions f_{I_i}, $i = 1, \ldots, n$, $m = 2$ machines with relative speeds $c_1 = 1$, $c_2 \geq 1$, and lifetime distributions $F_{S_j}(t) = \Pr(S_j < t)$, $j = 1, 2$, compute $\Pr(Z \leq t)$, where Z is a random variable describing the time of successful completion of these n routines on these two machines under eager scheduling.*

In this Section 5.4.2, machines are assumed to be perfectly reliable; a restriction that is removed for the worker machines in the following Section 5.4.3 and for the master machine in Section 5.4.5.

As we have seen above, the approach, based on the law of total probability, for the solution is to consider all possible combinations of routine runtimes, compute the probability that such a combination is successfully executed before time t, and sum up all these probabilities, weighted by the probabilities of a given combination actually occurring:

$$\Pr(Z \leq t) = \int_{\vec{x} \in (\mathbb{R}_0^+)^n} \Pr(\vec{x} \text{ occurs}) \\ \Pr(\vec{x} \text{ successfully completed before } t \mid \vec{x} \text{ occurrs}) \ \mathrm{d}\vec{x} \tag{5.1}$$

The proof will proceed by identifying subsets of $(\mathbb{R}_0^+)^n$ (the n-fold Cartesian product of the set of non-negative real numbers) such that all combinations of routine runtimes in such a subset show the same behavior under eager scheduling, and then computing the success probabilities for all combinations in each subset.

In the following, individual routines are indicated by i, $i = 1, \ldots, n$, and machines are denoted by j, $j = 1, \ldots, m$. k, l, u, and v are general variables. x_i represents the execution time of routine i on a machine of speed 1. Tuples of fixed length k are written as $(y_1, y_2, \ldots, y_k)$, and ordered, variable-length sequences as $[y_1, y_2, \ldots]$. If A is a tuple or sequence, $|A|$ is the number of elements in it, and $A(l)$ is element l of A if $1 \leq l \leq |A|$. Sequences can be concatenated with the operator $\circ$.

Two auxiliary structures will be used in the proof—assignment sequences A and lists of inequalities L.

Definition 5.1 (Assignment sequence A). *An* assignment sequence *A is a sequence of pairs, written as $[(j_1, i_1), \ldots, (j_k, i_k)]$, where j_l indicates a machine and i_l is a routine, $1 \leq l \leq |A|$.*

A is used to represent assignments of routines to machines, along with their *relative*, not absolute, finishing times. For example, $A = [(2,2),(1,1),(2,3)]$ means that first Routine 2 finishes on Machine 2, then Routine 1 on Machine 1, and finally Routine 3 finishes on Machine 2.[3]

[3] Such an assignment sequence is the formal counterpart to a block in Figure 5.1.

Definition 5.2 (List of inequalities L). *A* list of inequalities *L is a sequence of pairs of inequalities.*

Inequalities are between linear combinations of routine runtimes. L is used to represent subsets of $(\mathbb{R}_0^+)^n$. Inequality lists are associated with assignment sequences: (L, A). Sets of such associations are denoted by $\mathcal{A}$. To conveniently construct inequalities from assignment sequences, the following utility function is used.

Definition 5.3 (Summing up assignments). *The* sum *of runtimes of routines assigned to a machine j in assignment A is*

$$sum(A, j) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } A = [] \\ x_i + sum(A', j) & \text{if } A = [(j, i)] \circ A' \\ sum(A', j) & \text{if } A = [(j', i')] \circ A' \text{ and } j' \neq j. \end{cases}$$

The weighted sum $wsum(A, j)$ *additionally takes the relative speeds of the machines into account:*

$$wsum(A, j) \stackrel{\text{def}}{=} \frac{1}{c_j} sum(A, j)$$

$wsum(A, j)$ is the time it takes for machine j to execute all the routines assigned to it in some given assignment A, provided no fault occurs.

We will now construct $2^{(n-1)}$ associations by induction:[4]

Definition 5.4 (Set of states $\mathcal{A}_k$). *$\mathcal{A}_k$, the set of all possible associations after $k \leq n$ scheduling steps, is defined as follows:*

Induction start: *Let $L_0 = [(0 < x_1, x_1 < c_2 t)]$, $A_0 = [(1, 1)]$, and $\mathcal{A}_0 = \{(L_0, A_0)\}$.*

Induction step $k \rightsquigarrow k + 1$: *Start with an empty $\mathcal{A}_{k+1}$. For every association (L, A) in $\mathcal{A}_k$, construct two new associations (L_1, A_1) and (L_2, A_2):*

- *(L_1, A_1): If $A = [(j_1, i_1), \ldots, (j_k, i_k)]$, $A_1 = [(j_1, i_1), \ldots, (3 - j_k, k + 1), (j_k, i_k)]$. $L_1 = L \circ L'$ where $L' = [(0 < x_{k+1}, wsum(A_1, 3 - j_k) < wsum(A_1, j_k))]$.*
- *(L_2, A_2): If $A = [(j_1, i_1), \ldots, (j_k, i_k)]$, $A_2 = [(j_1, i_1), \ldots, (j_k, i_k), (3 - j_k, k + 1)]$. $L_2 = L \circ L'$ where $L' = [(wsum(A_2, j_k) < wsum(A_2, 3 - j_k), x_{k+1} < c_2 t)]$.*

Add (L_1, A_1) and (L_2, A_2) to $\mathcal{A}_{k+1}$.

We start the proof of the correctness of this definition with two simple observations.

Lemma 5.1. *In any assignment sequence generated by eager scheduling on two machines, the machine that becomes idle last is not the machine that becomes idle first.*

Proof. Obvious, as long as both machines are operating. □

Lemma 5.2 (Machines are not left idle.). *While there are routines left that are not executed, the following situation cannot occur under eager scheduling: One machine finishes a routine, and remains idle, while the other machine finishes two routines.*

Proof. Suppose Machine 1 finishes one routine, and remains idle. Afterwards, Machine 2 finishes another routine, and yet another after it. This implies that Machine 2 is assigned a routine that has not been given to Machine 1, although Machine 1 has been idle. Contradiction. □

Indeed, this is mirrored in the structure of the associations constructed by Definition 5.4:

Lemma 5.3. *In any assignment generated by Definition 5.4 with length larger than 1, the last two routines are assigned to two different machines.*

[4] In this construction, note that $3 - j$ maps 1 to 2 and 2 to 1, effectively computing the "other" machine, since machines are numbered 1 and 2.

Proof. By induction on k, the number of scheduling steps.

Induction start $k = 2$**:** The first induction step of Definition 5.4 generates $A_1 = [(2,2),(1,1)]$ and $A_2 = [(1,1),(2,2)]$, which is correct.

Induction step $k \rightsquigarrow k+1$**:** The new assignment generated in the induction step goes to machine $3 - j_k \neq j_k$ and either in the last or the second-last place, which proofs the hypothesis.

□

We can now proof one of the main lemmas:

Lemma 5.4 ($\mathcal{A}_k$ correctly describes the first k scheduling steps.). *When using ordered eager scheduling, no other execution sequences expect those described by $\mathcal{A}_k$ occur during the first k steps, if at least $k \leq n$ routines are to be scheduled. Moreover, the list of inequalities associated with every assignment describe precisely the set of runtime combinations under which any given assignment occurs.*

Proof. By induction on k.

Induction start $k = 1$**:** Routine 1 will be placed on empty Machine 1, creating the assignment $A_0 = [(1,1)]$. This routine can only be completed by either machine if its execution takes at most $c_2 t$ time units. Hence it is sufficient to consider only such instances of Routine 1 that are between 0 and $c_2 t$ in length. Therefore, $\mathcal{A}_0 = \{([(0 < x_1, x_1 < ct)], [(1,1)])\}$ correctly describes all possible sequences with one scheduling step and all the possible values of x_1 that are relevant.

The case $k = 2$ is similar.

Induction step $k \rightsquigarrow k+1, k \geq 2$**:** The scheduling step $k+1$ can start from any association in $\mathcal{A}_k$, but no other (by induction hypothesis). Given any (L, A) in $\mathcal{A}_k$, we have to show that Definition 5.4 correctly generates all possible other scenarios.

In step $k+1$, routine $k+1$ will be selected for assignment by the scheduler. This routine will have an execution time of x_{k+1} (unknown to the scheduler, but nevertheless existing). It will be assigned once either of the two machines becomes idle. In an assignment sequence $A = [(j_1, i_1), \ldots, (j_k, i_k)]$, this cannot be machine j_k, since it becomes idle last; it must be the other machine (Lemma 5.1). The machine that becomes idle first is therefore machine $3 - j_k$.

This machine will therefore be assigned routine $k+1$, which terminates either before or after routine i_k terminates on the other machine. We also know (because of Lemma 5.2) that the second-to-last finishing time in A must be on machine $3 - j_k$. Hence only two new assignments of relative finishing times are possible: $A_1 = [(j_1, i_1), \ldots, (3 - j_k, k+1), (j_k, i_k)]$ if the new routine finishes before routine i_k, or $A_2 = [(j_1, i_1), \ldots, (j_k, i_k), (3 - j_k, k+1)]$, if it finishes last.

When do these two cases occur? It depends on the length of the new routine $k+1$. Since all the execution times $x_1, \ldots, x_k$ of routines are already bounded by L, this generates an additional restriction on the length of routine $k+1$, in addition to the two trivial restrictions $0 < x_{k+1}$ and $x_{k+1} < c_2 t$, which hold for any routine.

The first case, routine $k+1$ finishing first, occurs if the total execution time of all routines on machine $3 - j_k$ is less than the total execution time of all routines on the other machine (after routine $k+1$ has been added): $wsum(A_1, 3 - j_k) < wsum(A_1, j_k)$, which gives an upper bound on x_{k+1}, strengthening $x_{k+1} < c_2 t$. This is precisely the new inequality generated by L_1 in Definition 5.4. The second case is symmetric and generates L_2, complementing A_2.

□

It is also easy to see from the proof of Lemma 5.4 that any association in $\mathcal{A}_k$ is a possible behavior of the eager scheduling algorithm.

The following Lemma 5.5 formulates the last observation in the proof of Lemma 5.4 more clearly.

Lemma 5.5 (Inequalities are well formed.). *For any (L, A) in $\mathcal{A}_k$, the uth pair of inequalities ($1 < u \leq k$) defines x_u only in terms of x_v, $1 \leq v < u$. Moreover, the two members of the uth pair supplement each other.*

Proof. By induction on k.

Induction start $k = 1$: Clear.

Induction step $k \rightsquigarrow k + 1$: Let $(L, A) \in \mathcal{A}_k$. Distinguish the two cases from Lemma 5.4.

1. $L_1 = [(0 < x_{k+1}, wsum(A_1, 3 - j_k) < wsum(A_1, j_k))]$. This implies that
$$\frac{x_{k+1}}{c_{(3-j_k)}} < wsum(A_1, j_k) - wsum(A, 3 - j_k).$$
2. $L' = [(wsum(A_2, 3 - j_k) > wsum(A_2, j_k), x_{k+1} < c_2 t)]$. This implies that
$$wsum(A_1, j_k) - wsum(A, 3 - j_k) < \frac{x_{k+1}}{c_{(3-j_k)}}.$$

The claim follows since routine $k + 1$ is not assigned to machine j_k in either A_1 or A_2.

□

Corollary 5.1. *x_i is bounded in an interval by L_i.*

We have now established a set of 2^{n-1} associations, where each association describes a subset of $(\mathbb{R}_0^+)^n$ (the n-fold cartesian product of the nonnegative real numbers) such that all vectors of routine runtimes $\vec{x}$ in such a subset have the same scheduling behavior under eager scheduling; their very behavior is described by the assignment component A of the association.

In this state of the execution, all routines have been assigned once. The next step taken by an eager scheduling algorithm is to eagerly assign one routine a second time to the first free machine—we are still assuming that no fault has occured. There is only one routine that has not finished when all routines have been assigned once and one machine becomes idle: the routine that is currently executing on the other, non-idle machine. In the notation of assignment sequences, it is the last routine in the sequence. And as always, there is no simple way of knowing in advance whether this eagerly scheduled routine will finish before or after its peer execution, so both cases have to be considered (see Figure 5.1 for an illustration).

Definition 5.5 ($\mathcal{A}_{\mathrm{ES}}$). *Given the association set $\mathcal{A}_n$, the association set after eagerly scheduling the remaining unfinished routine is called $\mathcal{A}_{\mathrm{ES}}$ and defined as follows:*

For each (L, A) in $\mathcal{A}_n$, where $A = [(j_1, i_1), \ldots, (j_n, i_n)]$, add the following two associations to $\mathcal{A}_{\mathrm{ES}}$:

1. *(L_1, A_1), where $A_1 = [(j_1, i_1), \ldots, (3 - j_n, i_n), (j_n, i_n)]$ and $L_1 = L \circ [wsum(A_1, 3 - j_n) < wsum(A_1, j_n), \cdot]$ (here, only one inequality is generated),*
2. *(L_2, A_2), where $A_2 = [(j_1, i_1), \ldots, (j_n, i_n), (3 - j_n, i_n)]$ and $L_2 = L \circ [wsum(A_2, 3 - j_n) > wsum(A_2, j_n), \cdot]$.*

Lemma 5.6 ($\mathcal{A}_{\mathrm{ES}}$ correctly describes eager scheduling without faults). *Under ordered eager scheduling, no execution sequences other than those described by $\mathcal{A}_{\mathrm{ES}}$ may occur, and the inequalities in $\mathcal{A}_{\mathrm{ES}}$ give the correct partitioning of the routine space.*

Proof. Follows from the previous paragraph. □

The additional constraint generated by eagerly scheduling one routine is not easily expressed as an interval limiting an area of integration. However, it can be expressed as an argument for a Heaviside function that is 1 if and only if the inequality is satisfied by any given combination of $x_1, \ldots, x_n$. We will use this formulation later on, with the shorthand $H(a < b) = H(b - a)$.

To complete the fault-free case, we need one additional utility function: given an assignment A, what is its completion time?

Definition 5.6 (Fault-free completion time). *Let A be an assignment sequence, and n the number of routines that is to be executed. Define the* completion index ci *as*[5]

$$\mathrm{ci}(A,n) \stackrel{\mathrm{def}}{=} \begin{cases} \mathrm{argmin}_{i,1\le i\le |A|}\, \rho(A[1..i]) = \{1,\ldots,n\} \\ \infty \textit{ if no such } i \textit{ exists} \end{cases}$$

where $A[1..i]$ is the prefix of A of length i and $\rho(A)$ is the set of routines occurring in A:

$$\rho(A) \stackrel{\mathrm{def}}{=} \begin{cases} \emptyset & \textit{if } A = [] \\ \{i\} \cup \rho(A') & \textit{if } A = [(j,i)] \circ A'. \end{cases}$$

The completion time of A *is defined as*

$$\mathrm{ct}(A) \stackrel{\mathrm{def}}{=} \begin{cases} wsum(A, A(\mathrm{ci}(A))(1)) & \textit{if } \mathrm{ci}(A) < \infty \\ +\infty & \textit{else} \end{cases}$$

($A(l)(1)$ is the machine parameter of the lth entry in A).

This function ct will be used to eliminate combinations of x_i that lead to completion times larger than t, again by means of a Heaviside function. This elimination is necessary even in the fault-free case since we always allowed all routine runtimes to range up to $c_2 t$, even if they are scheduled on the first machine.

Now the following Theorem 5.1 is complete and a simple rewriting of Equation (5.1).

Theorem 5.1 (Runtime distribution of eager scheduling, fault-free case). *For n routines with runtime distributions f_{I_i} and two fault-free worker machines with relative speeds $c_1 = 1, c_1 \le c_2$, the runtime distribution of the successful completion time $\Pr(Z \le t)$ of eager scheduling is:*

$$\Pr(Z \le t) = \sum_{(L,A)\in\mathcal{A}_{\mathrm{ES}}} \int_{L(1)} \cdots \int_{L(n)} f_{I_1}(x_1)\cdots f_{I_n}(x_n) H(L(n+1)) H(t - \mathrm{ct}(A)) \mathrm{d}x_n \ldots \mathrm{d}x_1$$

where the x_i's in both Heaviside arguments are interpreted as the running variables of the integrals.

Proof. By Lemma 5.6, $\mathcal{A}_{\mathrm{ES}}$ is a correct partitioning of the space of possible routine runtimes, and the inequalities in $\mathcal{A}_{\mathrm{ES}}$ identify regions of identical behavior. These regions are expressed by the bounds of the integrals, where $L(k)$ limits x_k in terms of x_l, $l = 1,\ldots,k-1$ for $k > 1$. The constraint stemming from the eager scheduling is expressed by $H(L(n+1))$, and the product $f_{I_1}(x_1)\cdots f_{I_n}(x_n)$ computes the occurrence probability of a given case. Any such case contributes to the probability of success before time t only if its completion time $\mathrm{ct}(A)$ is smaller than t or, equivalently, if $H(t - \mathrm{ct}(A)) \neq 0$. □

5.4.3 General solution for two potentially failing machines

Let us now proceed to analyzing the distribution with potentially faulty machines. Recall that machines are only allowed to crash, but not to, e.g., produce wrong results. Even under this assumption, however, there is a positive probability that a parallel step cannot be completed, since all machines could fail. This implies that $\lim_{t\to\infty} \Pr(Z \le t) < 1$. Therefore, it is—strictly speaking—incorrect to talk about the runtime distribution of Z. This term is nonetheless used as a convenient figure of speech.

Consider the behavior of eager scheduling if one of the machines crashes. As eager scheduling does not use any explicit fault detectors, a failing machine is transparently compensated for by the other one. In particular, for the runtime distribution, it does no longer matter in which order routines are scheduled on the surviving machine, since it has to execute all of them anyway. Note that after a fault, an assignment A does no longer necessarily correspond to the real schedule, but this is irrelevant with respect to the probability distribution. For convenience of proof, all assignments A are therefore extended so that all routines are assigned to both machines. This extension is merely a technicality to allow an easier notation during the proof and does not (and does not have to) reflect the actual routine executions.

[5] $\mathrm{argmin}_i\, p(i)$, where p is a predicate depending on i, is the smallest value of i such that $p(i)$ is true.

Definition 5.7 (Extended association set $\mathcal{A}_{\mathrm{F}}$). *Given an association set $\mathcal{A}_{\mathrm{ES}}$, the extended association set $\mathcal{A}_{\mathrm{F}}$ is defined as follows.*[6]

Start with an empty $\mathcal{A}_{\mathrm{F}}$. For each (L, A) in $\mathcal{A}_{\mathrm{ES}}$, add the association $(L, A \circ e(A))$ to $\mathcal{A}_{\mathrm{F}}$, where $e(\cdot)$ is[7]

$$e(A) \stackrel{\mathrm{def}}{=} \begin{cases} [] & \text{if } A = [] \\ e(A'') & \text{if } A = A' \circ [j, i], \\ & \text{routine } i \text{ appears in } A', \\ & \text{and } A'' \text{ is } A' \text{ with every} \\ & \text{occurrence of } i \text{ removed.} \\ [(3-j, i)] \circ e(A') & \text{else } (A = A' \circ [(j, i)]). \end{cases}$$

In the faulty case, it is no longer sufficient to simply check whether or not a given combination of x_i terminates before t or not. Rather, all possible fault scenarios have to be considered, their probability must be computed, and only if the execution of all routines terminates before t can this probability be added to the runtime distribution. This is a straightforward generalization of the term $H(t - \mathrm{ct}(A))$ of the fault-free case, where this only possible scenario occurs with probability 1. To solve this, we must first characterize fault scenarios S and their likelihood, and then compute the termination time of a given assignment A under a fault scenario S.

Definition 5.8 (Fault scenarios). *For scheduling n routines on two machines, the set of fault scenarios $\mathcal{S}_n$ is defined as*

$$\mathcal{S}_n \stackrel{\mathrm{def}}{=} \{(s_1, s_2) | s_1, s_2 \in \mathbb{N}_0 \wedge 0 \leq s_1, s_2 \leq n \wedge s_1 + s_2 \geq n\},$$

where s_j indicates the number of routines that machine j survives.[8]

Lemma 5.7 (Fault scenarios are sufficient). *For scheduling n routines on two machines, it suffices to consider only the set of fault scenarios as defined by Definition 5.8.*

Proof. To execute n routines, at least n routine assignments must be survived, implying $s_1 + s_2 \geq n$. At most n routines are assigned to each machine, which means that it is sufficient to only consider $s_i \leq n$, $i = 1, 2$. □

Lemma 5.8 (Fault scenarios are successful). *For any given assignment A, $(L, A) \in \mathcal{A}_{\mathrm{F}}$ and any fault scenario S defined according to Definition 5.8, all routines can be successfully executed.*

Proof. Let b_j be the number of routines assigned to machine j in A before the eager scheduling step was constructed (according to Definition 5.4), and note that $b_1 + b_2 = n$. Distinguish three cases:

1. $s_i \geq b_i$, $i = 1, 2$. Then both machines survive their normal assignments, and this fault scenario succeeds.

2. $s_1 < b_1$, $s_2 \geq b_2$. Since $s_2 - b_2 \geq b_1 - s_1$, Machine 2 survives enough routines to compensate for the failed Machine 1's inability to execute its last $b_1 - s_1$ routines. And eager scheduling will assign the unexecuted routines in some order to Machine 2. This eager assignment is reflected by the reverse order in which these routines appear in the assignments of $\mathcal{A}_{\mathrm{F}}$ in Definition 5.7.

   $s_1 \geq b_1$, $s_2 < b_2$ is completely analogous.

3. $s_l < b_l$, $l = 1, 2$ (neither machine surives its assignments) contradicts $s_1 + s_2 \geq n$ and is therefore no valid fault scenario.

[6] Note that in the following definition, L is not changed, since the pseudo-assignments of routines do not generate additional constraints on the relative runtimes of routines.

[7] The complicated second case is necessary since one routine has already been scheduled twice by the eager scheduling step; this routine must not be rescheduled again.

[8] I.e., $s_2 = 1$ means machine two survives the first routine, but fails during the execution of its second routine.

□

Since machines can fail, the time they need to execute only some of their assigned routines is of interest. This time is expressed by a slightly extended form of function $wsum$ that can be restricted to prefixes of assignment sequences:

Definition 5.9 (Weighted sum for prefixes). *The sum of routines in assignment A on machine j, restricted to prefix s, is defined as*

$$sum(A, j, s) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } A = [] \vee s = 0 \\ x_i + sum(A', j, s-1) & \text{if } A = [(j,i)] \circ A' \wedge s > 0 \\ sum(A', j, s) & \text{if } A = [(j',i)] \circ A' \wedge j' \neq j \end{cases}$$

and, analogous to Definition 5.3,

$$wsum(A, j, s) \stackrel{\text{def}}{=} \frac{1}{c_j} sum(A, j, s).$$

Definition 5.10 (Probabilities of fault scenarios). *For any given $A \in \mathcal{A}_{\mathrm{F}}$ and any fault scenario $S = (s_1, s_2) \in \mathcal{S}_n$, the occurrence probability of this fault scenario is given by*

$$\mathrm{pf}(A, S) \stackrel{\text{def}}{=} \mathrm{pf}_1(A, s_1)\mathrm{pf}_2(A, s_2)$$

where

$$\mathrm{pf}_j(A, s) \stackrel{\text{def}}{=} \begin{cases} \Pr(S_j < wsum(A, j, 1)) & \text{if } s = 0 \\ \Pr(wsum(A, j, s) < S_j < wsum(A, j, s+1)) & \text{if } 0 < s < n \\ \Pr(wsum(A, j, n) < S_j) & \text{if } s \geq n \end{cases}$$

for $j = 1, 2$.

Lemma 5.9 (Probabilities of fault scenarios are correct). *The probabilities for fault scenarios as defined by Definition 5.10 are correct.*

Proof. Faults on different machines are assumed to be independent, hence their probabilities can be computed separately and then multiplied. The probability of surviving the first s routines and failing in the $(s+1)$th routine, or at any time after the last routine if $s = n$, depends on the lengths of these routines assigned to the machine. This length is computed by $wsum$ as defined in Definition 5.9.

It does not make any difference if a routine's execution fails right after it started or at any other time. Therefore, it is sufficient to consider the probabilities as defined by function pf_j. □

As a last extension to the fault-free case, we need to consider the termination time of an assignment A under a given fault scenario S.

Definition 5.11 (Completion time under faults). *Let A be an assignment, n the number of routines to be executed, and $S = (s_1, s_2) \in \mathcal{S}_n$ a fault scenario. The* completion time under faults $\mathrm{ctf}(A, S)$ *is the time it takes to execute n routines with assignment A if fault scenario S occurs:*

$$\mathrm{ctf}(A, S) \stackrel{\text{def}}{=} \mathrm{ct}(\mathrm{prune}(A, s_1, s_2, 0, 0))$$

where prune *cuts all routines out of A that only happen after the faults occur:*

$$\mathrm{prune}(A, s_1, s_2, i_1, i_2) \stackrel{\text{def}}{=}$$

$$\begin{cases} [] & \text{if } A = [] \\ [(1,i)] \circ \mathrm{prune}(A', s_1, s_2, i_1 + 1, i_2) & \text{if } A = [(1,i)] \circ A' \wedge i_1 < s_1 \\ \mathrm{prune}(A', s_1, s_2, i_1 + 1, i_2) & \text{if } A = [(1,i)] \circ A' \wedge i_1 \geq s_1 \\ [(2,i)] \circ \mathrm{prune}(A', s_1, s_2, i_1, i_2 + 1) & \text{if } A = [(2,i)] \circ A' \wedge i_2 < s_2 \\ \mathrm{prune}(A', s_1, s_2, i_1, i_2 + 1) & \text{if } A = [(2,i)] \circ A' \wedge i_2 \geq s_2 \end{cases}$$

We can now finally formulate and proof our main theorem.

Theorem 5.2 (Runtime distribution of eager scheduling with possibly faulty machines). *For n routines with runtime densities f_{I_i}, two machines with relative speeds $c_1 = 1$, $c_2 \geq c_1$, and machine reliability as described by survival time S_j, the runtime distribution of eager scheduling is:*

$$\Pr(Z \leq t) = \sum_{(L,A)\in\mathcal{A}_{\mathrm{ES}}} \int_{L(1)} \cdots \int_{L(n)} f_{I_1}(x_1) \cdots f_{I_n}(x_n) H(L(n+1)) \\ (\sum_{S\in\mathcal{S}_n} \mathrm{pf}(A,S) H(t - \mathrm{ctf}(A,S))) \mathrm{d}x_n \ldots \mathrm{d}x_1 \tag{5.2}$$

Proof. The argument is similar to the proof of Theorem 5.1. Instead of only considering the case "no faults" with an assumed probability 1 and contributing to the distribution only if $H(t - \mathrm{ct}(A)) > 0$, we now have to consider all relevant fault scenarios. Lemma 5.8 guarantees that only the fault scenarios of $\mathcal{S}_n$ have to be considered and Lemma 5.9 shows how to compute the probabilities of the occurrence of such a scenario. Since all scenarios in $\mathcal{S}_n$ are disjoint, their probabilities can simply be added. A scenario succeeds before time t and adds to the distribution only if $t \leq \mathrm{ctf}(A,S) \Leftrightarrow H(t - \mathrm{ctf}(A,S)) > 0$. Adding up these successful fault scenarios for a given combination of x_i gives the probability of succeeding for this combination. □

5.4.4 Solution for m machines and routines with fixed runtimes

Unfortunately, the solution derived in Section 5.4.3 is computationally very expensive. It is difficult to compute the probability distribution in reasonable time with such general assumptions as have been made above. However, it might be more feasible to do so if the assumptions are more restricted. In this section, fixed runtimes of routines are assumed, i.e., routine i has a precisely given runtime a_i (on a machine of relative speed one).

These restrictions allow a more straightforward solution, which can be generalized to m machines right away. Much of the notation introduced in the previous section will be used here, with obvious generalizations from two machines to m machines where necessary. To avoid some purely technical special cases in the following derivation, we will assume that $n > m$.

The proof will use the notion of a state:

Definition 5.12 (State q of an execution). *A state $q = (A, S, R) \in \mathcal{Q}$ represents the progress of the execution of n routines on m machines, $\mathcal{Q}$ is the set of all possible states. Such a state consists of*

- *an assignment A similar to Definition 5.1, but each element extended by the absolute time of completion: $A = [(\text{machine } j, \text{routine } i, \text{absolute completion time } o), \ldots]$,*
- *a fault scenario S as in Definition 5.8, extended to m machines,*
- *a tuple R of length n that represents the state of each individual routine; an element of R can be either* waiting, started, *or* finished.

Note that S_j is the random variable corresponding to the lifetime of machine j, while s_j is the number of routines that machine j survives in a given state. Again we need some utility functions to conveniently formulate the definitions and proofs.

Definition 5.13 (The done predicate). *Given a state $q = (A, S, R)$,* done *is defined as*

$$\mathrm{done} \stackrel{\mathrm{def}}{=} \begin{cases} \mathcal{Q} \to \{\mathrm{true}, \mathrm{false}\} \\ q \mapsto \forall i \in \{1, \ldots, n\} : R(i) = \mathrm{finished}. \end{cases}$$

Definition 5.14 (The number of operational machines in a state). *For a state $q = (A, S, R)$, the number of operational machines in a state is given by*

$$\text{operational} \stackrel{\text{def}}{=} \begin{cases} \mathcal{Q} \to \mathbb{N}_0 \\ q \mapsto |\{j : S(j) \geq n\}|. \end{cases}$$

Definition 5.15 (The next routine to be scheduled). *Given a state $q = (A, S, R)$ that is not yet done, the next routine to be scheduled is given by*

$$\text{candidate} \stackrel{\text{def}}{=} \begin{cases} \mathbb{N}_0^n \to \mathbb{N}_0 \\ R \mapsto \begin{cases} \text{argmin}_i\, R(i) = \text{waiting} & \textit{if such an } i \textit{ exists,} \\ \text{argmin}_i\, R(i) = \text{started} & \textit{else.} \end{cases} \end{cases}$$

The assignments here are ordered according to absolute completion time. An ordered concatenation operator allows to maintain this order when assignments are concatenated, and only assignements ordered according to completion are used in this proof.

Definition 5.16 (Ordered concatenation of assignments.). *Given an assignment sequence A ordered by completion time, the* ordered concatenation *of A with a single assignment (j, i, c) is defined as*

$$A \star (j, i, c) \stackrel{\text{def}}{=} A_1 \circ [(j, i, c)] \circ A_2$$

where $A = A_1 \circ A_2$, $\forall k \in \{1, \ldots, |A_1|\} : A_1(k)(3) \leq c$, $\forall k \in \{1, \ldots, |A_2|\} : A_2(k)(3) > c$, and $\circ$ is normal concatenation.

Slightly more complicated—due to the need to consider faults and the startup phase of the scheduling algorithm—is the problem of determining the first machine that becomes idle in any given state q.

Definition 5.17 (Idle information (j, l, i, o)). *Given a state $q = (A, S, R)$, j (the first machine to become idle), its index l in A, the routine i it has been executing, and its absolute completion time o, are defined as follows:*

If $|\{u : R(u) = \text{waiting}\}| > n - m$ (not all machines have been assigned routines in this parallel step, and no routine has actually finished yet),

$$j = l = |\{u : R(u) = \text{started}\}| + 1 \textit{ and } o = i = 0$$

and else (all machines have been assigned routines, and some routine actually finishes),

$$l = \text{idleindex}(A, S, \vec{0}), j = A(l)(1), i = A(l)(2), o = A(l)(3)$$

where

$$\text{idleindex}(A, S, J) \stackrel{\text{def}}{=} \begin{cases} \infty & \textit{if } A = [], \\ 1 & \textit{if } A = [(j_1, i_1, o_1)] \circ A', \\ & j_1 \textit{ does not appear again in } A', \\ & \textit{and } S(j_1) \geq J(j_1), \\ 1 + \text{idleindex}(A', S, & \textit{else } (A = [(j_1, i_1, o_1)] \circ A'). \\ \qquad J \leftarrow (j_1, J(j_1) + 1)) & \end{cases}$$

Here J is a tuple of length m, representing the number of routines that have been assigned to each machine, and $(J \leftarrow (k, l))(u) \stackrel{\text{def}}{=} \begin{cases} J(u) & \textit{if } u \neq k, \\ l & \textit{else.} \end{cases}$

Lemma 5.10 (The idle information (j, l, i, o) is correct). *Given a state $q = (A, S, R)$, the idle information (j, l, i, o) as defined in Definition 5.17 correctly describes j, the next machine to become idle, its index l in A, the routine i that it has executed (if any), and the absolut completion when this machine becomes idle. $i = 0$ and $o = 0$ indicate that not all machines have been assigned a routine in this parallel step (it corresponds to the start phase of the parallel step).*

Proof. While there are more than $n - m$ routines waiting, less than m machines have been assigned a routine. This happens at the beginning of a parallel step, when no routine is finished yet. Hence the next idle machine is given by the number of routines started so far, plus 1.

Otherwise, the first machine to become idle is the machine that appears first in the assignment without any other routines being assigned to it later, where faults have to be taken into account. Faults are represented by the number of routines a machine survives, so the m-tuple J counts this number for each machine. If $S(j) \geq J(j)$, machine j has survived all previous and the current routine, so if it is the last routine assigned to this machine, then the machine is idle and still working. The first machine for which this holds is the first idle machine (since assignement A is ordered by completion time).

If all machines fail before they have completed their final routine (and only then) will the value of the $\mathrm{idleindex}$ function be infinity. Assignments have to assure that at least one machine survives until completion of a parallel step. □

We can now construct a function that "executes" the scheduling of a routine by mapping a state q to two succeeding states q_1 and q_2, where q_1 represents the normal progress of the computation and q_2 represents the event that the machine on which the routine is scheduled fails during the execution of this routine.

Definition 5.18 (Eager scheduling on states). *Given a state $q = (A, S, R)$, the function* es *is*

$$\mathrm{es} \stackrel{\mathrm{def}}{=} \begin{cases} \mathcal{Q} \to 2^{\mathcal{Q}} \\ q \mapsto \begin{cases} \{q\} & \textit{if } \mathrm{done}(q) \\ \{q_1, q_2\} & \textit{else,} \end{cases} \end{cases}$$

where q_1, q_2 are defined as follows.

If $\mathrm{done}(q)$ holds, $\mathrm{es}(q) = \{q\}$. Let (j, l, i, o) be the idle information of state q as defined by Definition 5.17. Then

$$R' = \begin{cases} R & \textit{if } i = 0 \\ R \leftarrow (i, \mathrm{finished}) & \textit{else} \end{cases}$$

*is the new routine state. If $\mathrm{done}(R')$ is true, $\mathrm{es}(q) = \{[A, S, R']\}$, $\mathrm{ctf}(q) = o$, and $\mathrm{ci}(q) = l$ (*ci *is the* completion index *to be used later). Otherwise, $i_{\mathrm{cand}} = \mathrm{candidate}(R')$ is the next routine to be scheduled. Then*

$$q_1 = [A \star (j, i_{\mathrm{cand}}, o + \frac{a_{i_{\mathrm{cand}}}}{c_j}), S, R' \leftarrow (i_{\mathrm{cand}}, \mathrm{started})]$$

and if $\mathrm{operational}(S) > 1$,

$$q_2 = [A \star (j, i_{\mathrm{cand}}, o + \frac{a_{i_{\mathrm{cand}}}}{c_j}), S \leftarrow (j, \mathrm{count}(A, j)), R' \leftarrow (i_{\mathrm{cand}}, \mathrm{started})],$$

else $q_2 = q_1$. Here $\mathrm{count}(A, j)$ is the number of routines that have been assigned to machine j in A.

Starting from an initial state $q_0 = ([], (n, \ldots, n), (\mathrm{waiting}, \ldots, \mathrm{waiting}))$, the repeated application of function es generates all possible execution scenarios by either letting a machine survive the assignment of a new routine (q_1), or by assuming that it crashes during this assignment (q_2). This is proven in the following lemmas.

Lemma 5.11 (es **reflects an eager scheduling step).**

Proof. Consider a state $q = (A, S, R)$. If in this state all routines are done, the algorithm terminates.

If there is at least one routine that is not done, the algorithm selects the first machine that is or becomes idle and assigns this routine to it. The completion time of this new routine is its execution time divided by the relative machine speed, plus the time at which the machine becomes idle (which can be 0).

For each assignment of a routine, there are two cases (as long as there are at least two operational machines): either the machine survives the execution of this routine, or it does not. In both cases, the assignment is added to A, but in the latter case, S is marked to indicate that this machine does not survive the execution of this routine. N.B. that the number of operational machines is not allowed to fall below 1. Therefore, the $\mathrm{idleindex}$ function will never be ∞.

□

Definition 5.19 (Set of all executions Q_{es}). *$Q_{\mathrm{es}} = \lim_{k\to\infty} \mathrm{es}^k(q_0)$, where the application of es to a set of states is defined per element, $\mathrm{es}(Q) = \cup_{q\in Q}\mathrm{es}(q)$, $Q \in 2^{\mathcal{Q}}$ and the initial state is $q_0 = ([], (n, \ldots, n), (\mathrm{waiting}, \ldots, \mathrm{waiting}))$.*

Lemma 5.12 (Q_{es} is finite). *Q_{es} is finite, and only a finite number of steps are necessary to generate it, i.e., there is a k_0 such that $\forall k \geq k_0 : \mathrm{es}^k(q_0) = \mathrm{es}^{k_0}(q_0)$.*

Proof. We have to show that for any state not done, there are only a finite number of successors. Think of the repeated application of es as a tree. Obviously, the tree is of finite degree.

First note that a state for which done holds is a fixpoint of es. Suppose there is an infinite path (q) in the tree, $q_{l+1} = \mathrm{es}(q_l)$ and $\forall l : q_{l+1} \neq q_l$. Such a path implies $\forall l : \mathrm{done}(q_l) = \mathrm{false}$. However, every application of es reduces the number of routines that are not started or not finished by one (since at least one machine must not fail), and this number cannot fall below zero. Contradiction.

Therefore, by König's Lemma, the tree is finite, and Q_{es} is finite. And since any state q for which $\mathrm{done}(q)$ holds is a fixpoint of es, a finite number of steps suffice to generate Q_{es}. □

Lemma 5.13 (Q_{es} reflects eager scheduling).

Proof. Follows immediately from Lemma 5.11 and Lemma 5.12: Every state in Q_{es} corresponds to an actual execution of eager scheduling and there are no other executions possible (by Lemma 5.11) since ordered eager scheduling behaves deterministically modulo faults, which are accounted for. □

We finally have to compute the probability of any $q \in Q_{\mathrm{es}}$ actually happening. Unlike the proof in Section 5.4.2 and Section 5.4.3, the only probabilistic element here are the machine faults. The occurring faults are described by the survival parameter S in a state $q = (A, S, R)$. The probability for each machine $1, \ldots, m$ behaving as prescribed by q is given below.

Definition 5.20 (Probability of state q). *Given a state $q = (A, S, R)$, the probability of machine $j \in \{1, \ldots, m\}$ behaving as in q is*

$$\mathrm{pf}_j(q) = \begin{cases} \Pr(S_j < wsum(A, j, 1)) & \text{if } S(j) = 0 \\ \Pr(wsum(A, j, S(j)) < S_j < wsum(A, j, S(j) + 1)) & \text{if } 0 < S(j) < n \\ \Pr(wsum(A, j, n) < S_j) & \text{if } S(j) = n \end{cases}$$

Lemma 5.14 (pf_j is correct). *The probability of a state as defined by Definition 5.20 is correct.*

Proof. Analogous to Lemma 5.10. □

Hence the final theorem of this section can be formulated as follows.

Theorem 5.3 (Runtime distribution with fixed routine execution times). *For n routines with fixed execution times a_i, $i = 1, \ldots, n$ on a processor of speed 1, and $m < n$ processors of relative speed c_j and lifetime S_j, $j = 1, \ldots, m$, the runtime distribution of the successful completion time $\Pr(Z \leq t)$ of eager scheduling is*

$$\Pr(Z \leq t) = \sum_{q=(A,S,R)\in Q_{\mathrm{es}}} (\prod_{j=1}^{m} \mathrm{pf}_j(q)) H(t - \mathrm{ctf}(A,S))$$

where ctf *is the completion time under faults as generalized from Definition 5.11.*

Proof. By Lemma 5.13, Q_{es} is the set of all possible successful executions of eager scheduling with the given parameters. By Lemma 5.20, the probability of such a state occurring is $\prod_{i=1}^{m} \mathrm{pf}_i(q)$ (owing to the independence assumption of machine failures). The state will be successfully completed before or at time t only if $H(t - \mathrm{ctf}(A,S)) \neq 0$. □

5.4.5 Faults in the master

As has already been pointed out in Section 4.2.2, the master process in Calypso is a single point of failure. More precisely, this process (in this model, the dedicated master machine) must survive the entire computation to ensure that the program terminates correctly. If the master's lifetime is given by the random variable S_{master}, than the runtime distribution of a Calypso program is quite simply

$$\Pr(\bar{Z} \leq t) = \int_{\tau=0}^{t} \Pr(S_{\mathrm{master}} > \tau) f_Z(\tau) \mathrm{d}\tau \tag{5.3}$$

where f_Z is the density function associated with the runtime distribution $\Pr(Z \leq t)$ and $\bar{Z}$ is the random variable representing the time to successful completion of eager scheduling where both master and worker machines are potentially faulty. Examples for the consequences of an unreliable master process can be found in Section 5.5.3

5.5 Some examples

5.5.1 General solution for two machines

The following examples illustrate the analysis for general distributions of routine runtimes on two possibly faulty machines as presented in Section 5.4.3. The distribution of routine runtimes is uniform, the lifetime distribution of the two worker machines is exponential, Machine 1 is of relative speed 1 and Machine 2 of relative speed 2, and the master machine does not fail.

Figure 5.2 and Figure 5.3 each show the results for a numerical solution of Equation (5.2) and a simulation of eager scheduling with 50,000 runs.[9] Analysis and simulation produce very close results; the difference is hardly visible. In Figure 5.2, the routine runtime distribution is uniform with parameters 1 and 3,[10] the lifetime of both machines is exponentially distributed with mean 25 time units. While this is unrealistically low, it is necessary to make the effects of eager scheduling's fault masking visible. In Figure 5.3, the runtime distribution is again uniform with parameters 0 and 4, the mean lifetime is 100 time units.

5.5.2 Solution for routines with fixed runtime

The results of Section 5.4.4 are here illustrated with a parallel step with the parameters $a_i = 2i + 3$, $i = 1, \ldots, n$, $c_j = j$, $j = 1, \ldots, m$, and an exponentially distributed lifetime with mean 100. The small mean lifetime was again chosen to make the effects of faults more prominent. Similar to Section 5.5.1, both

[9] This large number of samples allows a very tight estimation of a confidence band around the empirical distribution following the Kolmogorov-Smirnov estimation: the true distribution is at most 0.6% different from the empirical one, at a 95% confidence level [103]. The analytical results are always within this confidence band, even though the numerical approximation suffers from the inherent problems of discretization. For an elaborate discussion of confidence bands, see Section 6.6.2, where it is particularly relevant owing to the smaller number of experiments there.

[10] Lower and upper boundary.

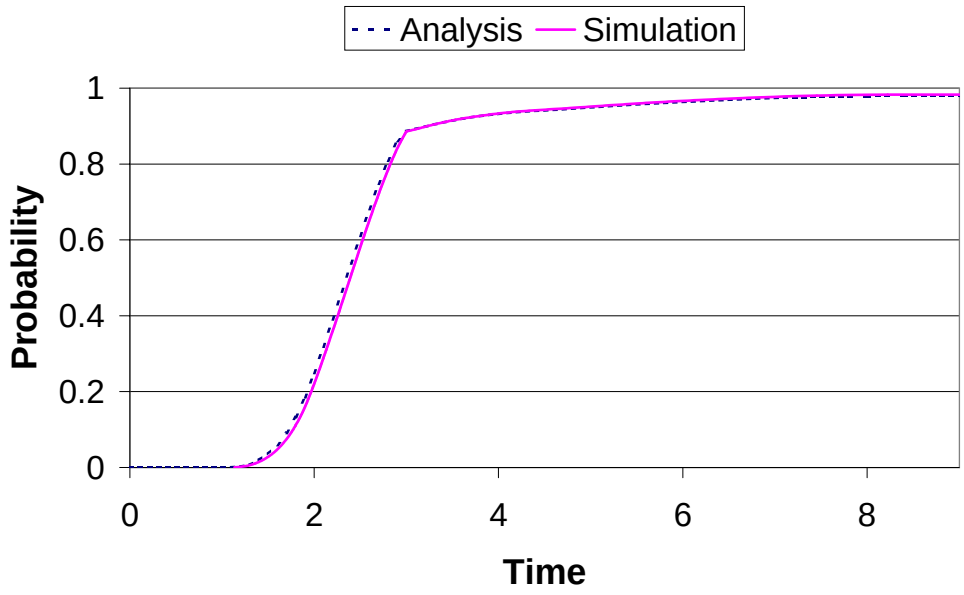

Figure 5.2: Runtime distribution of eager scheduling with $n = 3$ routines on $m = 2$ worker machines. Routine runtime is distributed according to $\mathcal{U}(1, 3)$, lifetime of both machines is exponentially distributed with mean 25, $c_2 = 2$.

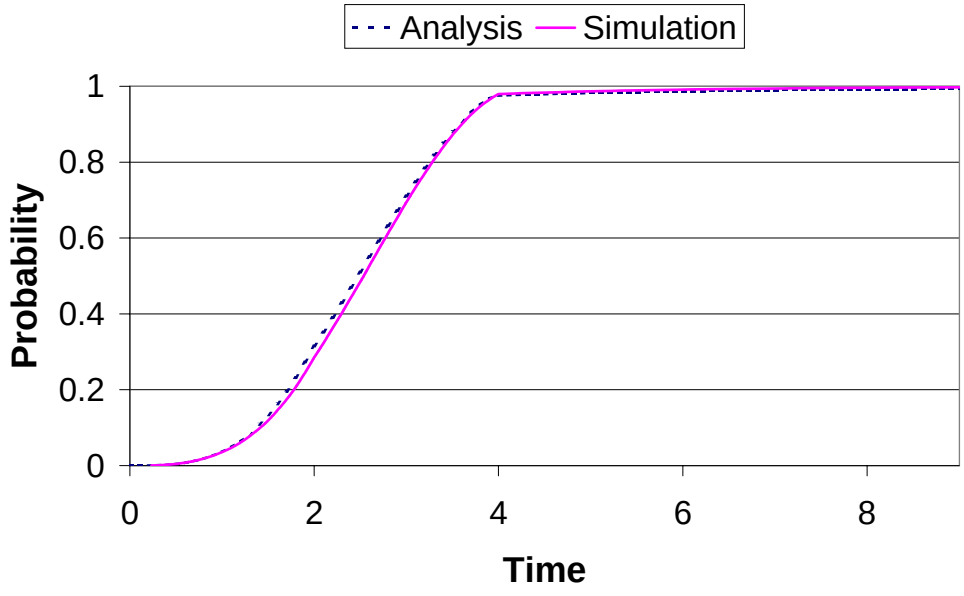

Figure 5.3: Runtime distribution of eager scheduling with $n = 3$ routines on $m = 2$ worker machines. Routine runtime is distributed according to $\mathcal{U}(0, 4)$, lifetime of both machines is exponentially distributed with mean 100, $c_2 = 2$.

analytical and simulative results are presented. In particular, Figure 5.4 shows the distribution for three machines executing six routines. The curves for simulation and analysis overlap almost precisely, so that only one line is actually visible. The time to compute this distribution analytically is slightly under two seconds on a Pentium 166—several orders of magnitude faster than the computation of the more general distribution for much smaller numbers of machines and routines in Section 5.5.1.

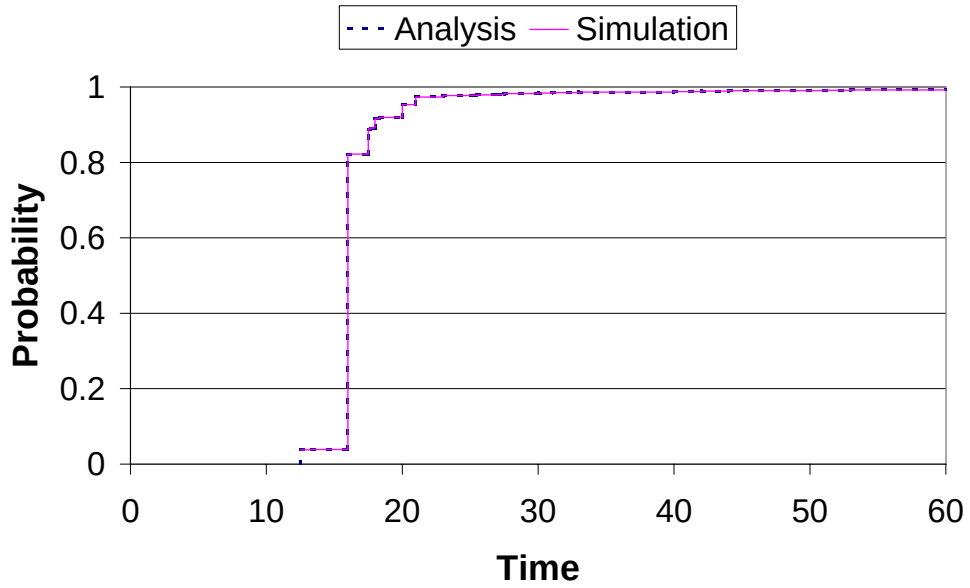

Figure 5.4: Runtime distribution of eager scheduling with $n = 6$ routines ($a_i = 2i + 3$) on $m = 3$ worker machines, lifetime of all worker machines exponentially distributed with mean 100, $c_j = j$.

A close inspection of this distribution function of Figure 5.4 shows unexpected behavior. There is a small probability (3.78 %) of completing the parallel step at time 12.5. This small probability is not commensurate with the algorithm's behavior under fault-free conditions. Indeed, the schedule of eager scheduling for this problem (without faults) is shown in Figure 5.5: it needs 16 time units to complete.

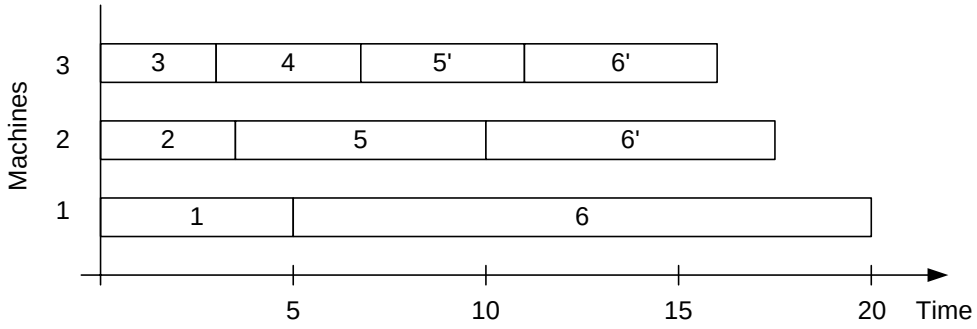

Figure 5.5: Schedule for $m = 3$, $n = 6$, task set $a_1 = 5$, $a_2 = 7$, $a_3 = 9$, $a_4 = 11$, $a_5 = 13$, $a_6 = 15$ ($a_i = 2i + 3$), $c_1 = 1$, $c_2 = 2$, $c_3 = 3$ ($c_j = j$), with all machines surviving.

An explanation for the completion time of 12.5 time units can be found in Figure 5.6. Here the first machine is assumed to have failed during its first routine. The crucial difference happens when Machine 3 asks for a job for the third time. In this scenario, Routine 6, the longest one, has not yet been assigned and is now given to the fastest machine. In the fault-free scenario, however, Machine 1 picks up Routine 6 after it has finished Routine 1 at time 5. Then, Machine 3 finds a situation in which all routines have been started.

Due to the specific variant of eager scheduling under consideration here, it picks Routine 5. As it turns out, this is a suboptimal decision, but note that even with simple heuristics such as "pick the routine that is already executing for the longest time" this same choice would have been made. The fact that eager scheduling does not find this assignment sequence shows that it is a non-optimal scheduling algorithm.

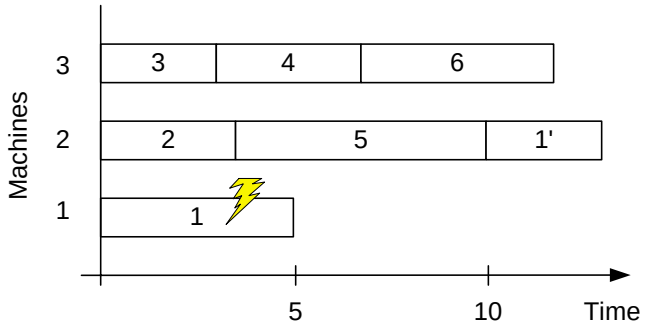

Figure 5.6: Schedule for $m = 3$, $n = 6$, task set $a_1 = 5$, $a_2 = 7$, $a_3 = 9$, $a_4 = 11$, $a_5 = 13$, $a_6 = 15$ ($a_i = 2i + 3$), $c_1 = 1$, $c_2 = 2$, $c_3 = 3$ ($c_j = j$), with machine 1 failing during its first step.

Finally, Figure 5.7 shows the runtime distribution for the same parameter settings, but with $m = 5$ machines executing $n = 20$ routines. Again note the excellent match of analytical and simulative results.

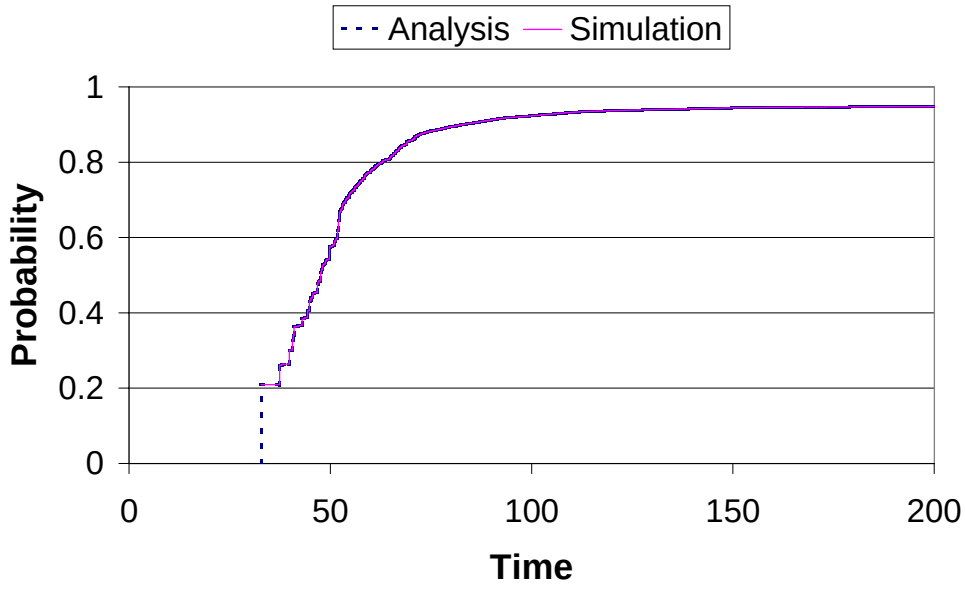

Figure 5.7: Runtime distribution of eager scheduling with $n = 20$ routines ($a_i = 2i + 3$) on $m = 5$ worker machines, lifetime of all worker machines exponentially distributed with mean 100, $c_j = j$.

5.5.3 Faults in the master

IIn the previous examples of Figures 5.4 and 5.7, the master machines has not been subjected to faults. Figure 5.8 shows the consequences of a master process running on an unreliable machine with all other parameters identical to Figure 5.4 and 5.7. Here, the master's machine, like all worker machines, has an exponentially distributed lifetime with mean 100 time units. The need to ameliorate this situation is evident: in the larger

example ($m = 5$,$n = 20$), the probability of eventually completing the program is reduced from 0.949 to 0.584.

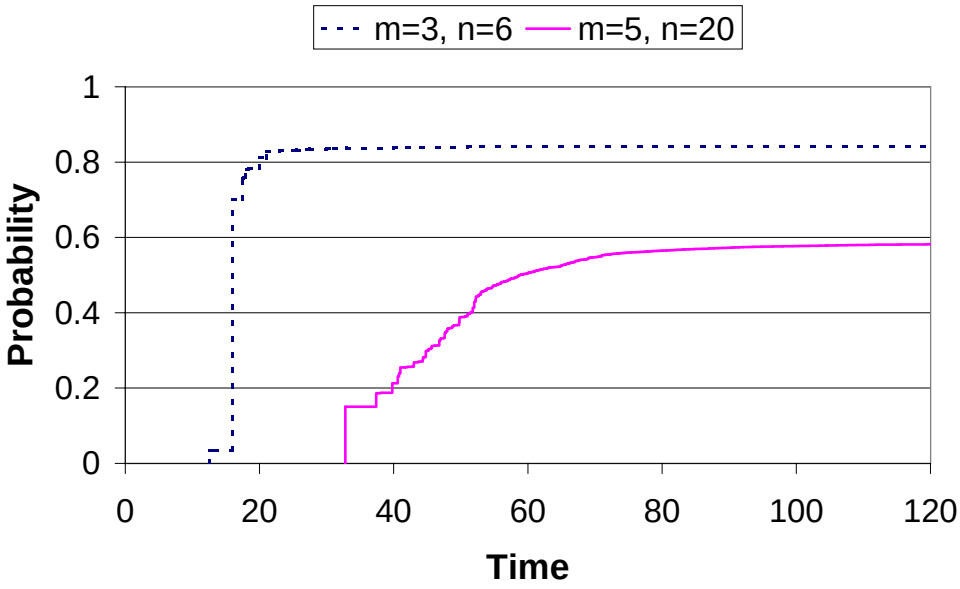

Figure 5.8: Runtime distribution for eager scheduling with unreliable master shown for $m = 3$,$n = 6$ and $m = 5$,$n = 20$, $a_i = 2i + 3$, $c_j = j$, lifetime of all worker machines exponentially distributed with mean 100.

5.6 Conclusions

An obvious conclusion to draw from this analysis is that it is necessary to limit the generality of one's assumptions to obtain manageable results. While in a purely theoretical sense, the more general the approach, the more satisfying the result, this is not necessarily true in this case. Even for simple configurations of two machines and three or four routines with general runtime distributions, computing the runtime distribution of the entire parallel step can take many days (due to the inherent necessity to compute nested integrals). On the other hand, the result for any given set of parameters is quite easily obtained by simulation.

This changes if the assumptions are more restricted. The analytical solution for routines with fixed runtimes is comparably fast to compute—on the order of seconds to a few minutes instead of days—a makes an analytical computation of the runtime distribution of eager scheduling quite feasible. Indeed, for the example of Figure 5.4, the analysis is faster than the simulation (for the simulation, a larger number of runs are necessary to closely match the analytical result).

5.7 Possible extensions

The analysis presented here suffers in its most general form from the high numerical complexity of the solution. It might prove interesting to consider advanced numerical techniques (e.g., Monte-Carlo approaches) to solve these integrals. However, such approaches might turn out to be very similar to simulations.

The solution can also be used to derive characteristic moments of the runtime distribution (such as average or standard deviation) for eager scheduling and compare it with other scheduling schemes. The analysis technique could also prove beneficial for investigations of other scheduling mechanisms.

An integration of these results in a resource management scheme such as the one described in [24], along with a simple description of the nature of the Calypso application (e.g., along the lines of the model discussed in Section 4.3), would allow the resource management system to make more informed decisions about the

consequences of adding or taking away resources from a particular program. Ideally, this description could include, along with information about the program itself and its deadline, precomputed values of the program's responsiveness with a varying number of resources. Similarly, tunable Calypso programs as described by CHANG et al. [49] could use this information to introspectively adapt their control flow to responsiveness requirements. Ideally, these two concepts should be integrated for maximum flexibility.

If I am given a formula, and I am ignorant of its meaning, it cannot teach me anything, but if I already know it what does the formula teach me?

– St. Augustine

Chapter 6

Checkpointing for Responsiveness

Checkpointing as a fault-tolerance mechanism for responsiveness is considered in this chapter. An analysis of the checkpointing interval problem is presented that optimizes the responsiveness of a service under given assumptions. This analysis is then used to derive checkpointing intervals for a Calypso version that checkpoints the master process.

6.1 Introduction

Any single point of failure like the master process of a Calypso program can be a serious obstacle to achieving high responsiveness. In this chapter, it is studied how checkpointing can be used to ameliorate this problem. Checkpointing is a widely used and well researched paradigm to improve fault tolerance. At certain times, the state of the system is written from volatile memory to stable memory (a checkpoint). In case a fault occurs (and is detected) a rollback recovery step takes place: The most recent checkpoint is restored back into main memory and the system resumes execution from this state.

Application areas for checkpointing are, e.g., transaction oriented database systems or long-running parallel applications [33]. Often, checkpointing optimization focuses on increasing the availability of a system or decreasing the mean response time of a service. Such optimizations are straightforward problem as long as no deadlines are considered (and a large body of knowledge is available related to this, see Section 6.2); adding deadlines and using responsiveness as evaluation metric makes this problem somewhat unusual, since it is no longer clear how often a checkpoint should be taken. The number of checkpoints to be taken during service execution is then the controlled parameter of an optimization problem, where service and system parameters (e.g., execution time, time to restart a process, or deadline) are given as independent variables.

As a basis for this optimization problem, a general model for a service under checkpointing is described in Section 6.3, along with an appropriate fault detection scheme. This fault detection is neither assumed to be immediate (which would not be realistic, in particular if mechanisms like a remote watchdog is used) nor perfect. In Section 6.4, an analysis of the problem of finding an optimal checkpointing interval for a service with a deadline is presented; some evaluations of this theoretical model are shown in Section 6.5. The theoretical analysis and evaluations for services with a fixed execution time has been performed in joint work with M. Werner; details can be found in [138, 139].

Adding checkpointing to Calypso, on the basis of this analysis, presents its own set of challenges. Checkpointing in parallel systems is usually complicated by the need to ensure consistency when distributed processes checkpoint their state. In Calypso, this is not the case: Only the master process has to write a checkpoint, since worker failures are handled by eager scheduling. Therefore, local checkpointing can be used, which makes checkpointing an even more attractive mechanism. Implementation issues of checkpointing a Calypso master are discussed in Section 6.6, where some experimental results are shown, too. The chapter is

concluded with Section 6.7 and possibilities for future work are outlined in Section 6.8.

6.2 Related work

Checkpointing in general is a widely researched area. A general overview of backward recovery methods can be found in [8]. In particular, the problem of choosing checkpointing intervals is of big practical importance and has received appropriate attention.

In a classic short paper, YOUNG [310] gives a first order approximation to the optimum checkpointing interval. The main limitation of this approach is that errors are not allowed to occur during error recovery. CHANDY et al. [48] investigate transaction-oriented systems with fixed or cyclically varying transaction request rates. Neither of these papers is concerned with real-time properties.

SHIN et al. [255] consider the problem of using checkpointing for real-time tasks if only imperfect fault detection mechanisms are available. Their optimization goal is the mean task execution time with the additional constraint that the probability of an unreliable result must be kept smaller than a prespecified value. They describe an algorithm for finding optimal placements of checkpoints to solve this problem. A particularly interesting result of this work is the fact that for imperfect fault detection mechanisms, equidistant checkpoints are only a suboptimal choice for their optimization goal. But since most available checkpointing packages (like the one presented in [298]) are based on equidistant checkpointing (unless checkpointing is directed by the programmer, usually with different considerations in mind), equidistant placement of checkpoints is a more realistic assumption. Furthermore, while SHIN et al. investigate real-time tasks, they ignore deadlines. As is shown later (see Section 6.5), the deadline does have a significant impact on the choice of an optimal checkpointing interval.

For performance reasons, many real checkpointing packages (like, e.g., [298]) typically do not wait for the completion of checkpoints. VAIDYA [286] uses Markov models to investigate the tradeoffs between checkpoint latency and overhead. For equidistant checkpoints, the optimal checkpointing interval is shown to be typically independent of the checkpoint latency. This result allows to ignore the impact of checkpointing latency in the following analysis. Moreover, in a real-time context, implementations with considerable checkpointing latency are usually undesirable since they can lead to substantial unpredictability.

KRISHNA et al. [154] acknowledge the need for evaluation criteria for real-time checkpointing other than mean execution time; they introduce a cost measure for checkpointing in a distributed system. They also provide a first approximation to an optimization that takes costs for both the user of the checkpointed service and other users of the system into account. Although this cost metric is more flexible than the responsiveness metric used here, it requires an application-dependent cost function, which can be undesirable.

The work most closely related to the analysis presented here is by GRASSI et al. [94]. They investigate both a system-oriented and service-oriented view of checkpointing and give a Laplace-Stieltjes transform of the probability distribution of the overhead caused by rollback recovery. The major difference to the approach presented here is that they consider immediate fault detection—the model described below only assumes detection by acceptance tests at discrete times—and they do not consider the problem of imperfect fault detection. Additionally, since their results are in Laplace transform, they are somewhat cumbersome to use—here results are obtained in the time domain and therefore do not require any inverse transformations. Similar arguments are true for the work by GEIST et al. [86]: While this paper arrives at very elegant solutions, it also does so by assuming immediate and perfect fault detection. Also, neither of these papers addresses the question of a stochastically described execution time.

For checkpointing in distributed systems, ELNOZAHY et al. [75] give an overview of the many possible methods. For the Calypso case, however, distributed checkpointing is not relevant since here only the single master process has to be checkpointed.

6.3 Model description

To analyze the checkpointing interval problem, a model for a service executing with checkpointing is needed. Using checkpointing breaks a service execution in a number of "blocks" (of equal length in the equidistant case). At the end of each block, a checkpoint is written that saves the current state to stable storage and an acceptance test is performed to check if any faults have occured during the execution of the previous block. Only if this test succeeds the results are deemed correct. Therefore, a checkpoint (or at least the acceptance test) is also performed after the very last block. Such an acceptance test covers computational faults in particular, but can be easily extended to cover crash faults (by means of a watchdog). Figure 6.1 shows service executions for $n = 1, 2, 3$ checkpoints. In case a fault is detected by the acceptance test, a recovery step is initiated and the most recent block is executed again.

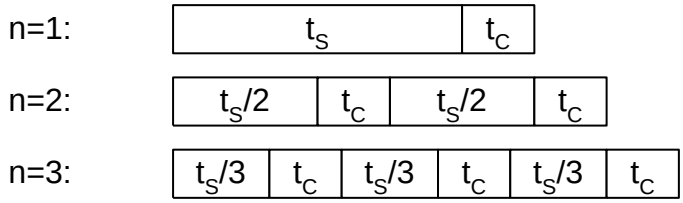

Figure 6.1: Fault-free checkpointing for different number of checkpoints n. Service execution time t_{S}, checkpointing time t_{C}.

The faults themselves are assumed to obey a Poisson process. The model does allow faults to occur during checkpointing or recovery, which means that the acceptance test can fail, too. In particular, it might not detect an existing fault—it can have imperfect coverage. However, a correct state will always pass the test.

The parameters are hence as follows:

t_{S} the exact service execution time of the service without any checkpointing.

t_{C} the (maximum) time required to take a checkpoint, assumed to be constant.[1] t_{C} also includes the time needed for the acceptance test.

t_{R} the time necessary to read in a checkpoint and establish the checkpointed state of a service after a fault has been detected. This is also assumed to be constant.

$n \geq 1$ the number of checkpoints (equidistantly) taken during fault-free service execution, i.e., the kth checkpoint is taken at time $k(t_{\mathrm{S}}/n) + (k-1)t_{\mathrm{C}}$ after the beginning of service execution—t_{S}/n is the checkpointing interval.

λ the (constant) fault rate of the Poisson process; $e^{-\lambda(t-t_0)}$ is the probability that the system works correctly in the interval $[t_0, t]$ provided it did so at time t_0.

p_{cov} the coverage probability of the acceptance test, i.e., the conditional probability that, given a fault occured during the execution of a block, it is detected by the following acceptance test.

This model is generalized by considering a service with a stochastically described execution time; the analysis for this case is presented in in Section 6.4.2. To do so, the fixed execution time t_{S} is replaced with a random variable I describing the service execution time without checkpointing (I for consistency with Chapter 5), and its cumulative density function F_I is used in the analysis.

This model has several limitations. In particular, the problem of distributed checkpointing is not addressed. However, since SILVA and SILVA [261] show that coordinated distributed checkpointing performs comparably

[1] While this is a common assumption for real-time checkpointing, it is not always made when checkpointing is used to optimize average execution time. ZIV and BRUCK [313], e.g., observe the state size of a program on-line and estimate the time it takes to write a checkpoint. An on-line algorithm for placing checkpoints is based on these estimations.

to independent checkpointing, it is conceivable that this analysis carries over to the distributed case, too. Furthermore, any restrictions imposed by the environment (e.g., the need to process sensor data at a given speed, which can put an upper bound on checkpointing intervals), are neglected. On the other hand, these restrictions can also be assumed to be expressed by the deadline itself. Another possible extension of the model would be a more fine-grained error detection (e.g., a watchdog interval shorter than $t_S/n + t_C$) or to integrate a restart mechanism after detecting a fault that has not been detected by an earlier detection check. The service is also assumed to be stationary in the sense of Definition 2.3—the main consequence is that λ has to be constant.

6.4 Analysis

6.4.1 Services with fixed execution time

To assess the responsiveness of a service, this analysis computes the probability distribution of the service's total execution time depending on t_S, t_C, t_R, n, λ, and p_{cov}. Let X_n be the random variable describing the service execution time and $F_{X_n}(t) = P(X_n \leq t)$ the distribution of the probability that the service is completed correctly before or at time t using n checkpoints.

Computing F_{X_n} directly is difficult.[2] As an intermediate step, compute the runtime distribution of a single block—these blocks are then recombined to derive the runtime distribution of the entire program. Consider first the case $p_{cov} = 1$ and let Y_n be the random variable describing the execution time of one of the service blocks of length t_S/n. A single block's execution cannot complete before $t_S/n + t_C$. If the system survives for at least $t_S/n + t_C$, this block is completed successfully—this case has probability $e^{-\lambda(\frac{t_S}{n}+t_C)}$. Therefore,

$$F_{Y_n}(t) = \begin{cases} 0 & \text{for } t < \frac{t_S}{n} + t_C \\ e^{-\lambda(\frac{t_S}{n}+t_C)} & \text{for } \frac{t_S}{n} + t_C \leq t < (\frac{t_S}{n} + t_C) + (t_R + \frac{t_S}{n} + t_C) \end{cases}$$

One additional recovery step takes another $t_R + t_S/n + t_C$ time units to complete before it increases the probability of successful execution. Executing a single recovery step happens when the system failed during the first execution of this block (with probability $(1 - e^{-\lambda(\frac{t_S}{n}+t_C)})$) but survived the execution of the recovery step (with probability $e^{-\lambda(t_R+\frac{t_S}{n}+t_C)}$):

$$F_{Y_n}(t) = \begin{cases} 0 & \text{for } t < \frac{t_S}{n} + t_C \\ e^{-\lambda(\frac{t_S}{n}+t_C)} & \text{for } \frac{t_S}{n} + t_C \leq t < (\frac{t_S}{n} + t_C) + (t_R + \frac{t_S}{n} + t_C) \\ (e^{-\lambda(\frac{t_S}{n}+t_C)} + & \text{for } (\frac{t_S}{n} + t_C) + (t_R + \frac{t_S}{n} + t_C) \leq t < \\ \quad (1 - e^{-\lambda(\frac{t_S}{n}+t_C)})e^{-\lambda(t_R+\frac{t_S}{n}+t_C)}) & \quad (\frac{t_S}{n} + t_C) + 2 * (t_R + \frac{t_S}{n} + t_C) \end{cases}$$

and so on for every additional recovery step. Obviously, the probabilities for surviving an ordinary step or a step with an additional recovery step are crucial. The following abbreviations will be used for these probabilities:

Definition 6.1 (ρ_1 and ρ_2).

$$\rho_1 \stackrel{\text{def}}{=} e^{-\lambda(\frac{t_S}{n}+t_C)}$$
$$\rho_2 \stackrel{\text{def}}{=} e^{-\lambda(t_R+\frac{t_S}{n}+t_C)}$$

[2] Except for the case of no checkpointing, for which the probability of successfully completing the program at time t_S is $e^{-\lambda t_S}$, and 0 before this time.

The probability distribution of Y_n is thus

$$F_{Y_n}(t) = \sum_{i=0}^{\infty} c_i H(t - a_i)$$

with

$$c_i = \begin{cases} \rho_1 & \text{for } i = 0 \\ (1-\rho_1)(1-\rho_2)^{i-1}\rho_2 & \text{for } i \geq 1 \end{cases}$$
$$a_i = (\frac{t_S}{n} + t_C) + i(t_R + \frac{t_S}{n} + t_C)$$

where $H(t)$ is the Heaviside function $H(t) = 0$ if $t < 0$, $H(t) = 1$ if $t \geq 0$. Note that this derivation holds only for a Poisson fault process. Assuming general fault processes, the occurrence of faults in previous steps must be considered when deriving the distribution.

This distribution is a so-called arithmetic distribution function [55]. Since there are, for any fixed set of parameters, only countably many points in time where Y_n changes its value, it is actually a discrete random variable. The discrete density function of Y_n is denoted by $f_{Y_n}(t)$; it is obtained from F_{Y_n} by replacing $H(\cdot)$ with the function $u(0) = 1$, $u(t) = 0$ if $t \neq 0$.

From this probability distribution F_{Y_n} it is now possible to compute the distribution for X_n. The total execution time of the service is the sum of the execution times for its n blocks: $X_n = \sum_{i=1}^{n} Y_n$. For (independent) random variables that have a density, the density of the sum of the random variables is the convolution of the densities. Therefore, the probability density of X_n is the n-fold convolution of the density of Y_n: $f_{X_n} = \underbrace{f_{Y_n} * \cdots * f_{Y_n}}_{n}$ ($*$ denotes convolution). Lemma 6.1 is helpful to compute this convolution.

Lemma 6.1 (Convolution). *Let* $f(t) = \sum_{i=0}^{\infty} c_i u(t - a_i)$. *Then*

$$(\underbrace{f * \cdots * f}_{n})(t) = \sum_{i_1,\ldots,i_n \in \mathbb{N}_0} (c_{i_1} \cdots c_{i_n}) u(t - (a_{i_1} + \cdots + a_{i_n}))$$

where the $a_{i_1} + \cdots + a_{i_n}$ *are not necessarily distinct.*

Proof. First proof the special case for the convolution $f^*(t)$ of two discrete functions $f_1(t)$ and $f_2(t)$ with $f_k(t) = \sum_{i=0}^{\infty} c_i^{(k)} u(t - a_i^{(k)})$, $k = 1, 2$

$$\begin{aligned} f^*(t) &= \sum_{k=0}^{t} f_1(k) f_2(t-k), t \in \mathbb{N}_0 \\ &= \sum_{k=0}^{t} \Big(\sum_{i=0}^{\infty} c_i^{(1)} u(k - a_i^{(1)}) \sum_{j=0}^{\infty} c_j^{(2)} u(t - k - a_j^{(2)}) \Big) \\ &= \sum_{i=0,j=0}^{\infty} c_i^{(1)} c_j^{(2)} \underbrace{\sum_{k=0}^{t} u(k - a_i^{(1)}) u(t - k - a_j^{(2)})}_{\phi(t)} \end{aligned}$$

with $\phi(t) = \begin{cases} 1 & \text{for } (k = a_i^{(1)}) \wedge (t - k = a_j^{(2)}) \\ 0 & \text{else} \end{cases}$ (all summands are 0 except for $k = a_i^{(1)}$). It is sufficient to consider only $t \geq 0$ since there are no negative execution times. Thus, $\phi(t) = 1 \Leftrightarrow t = a_i^{(1)} + a_j^{(2)}$ implies

$$f^*(t) = \sum_{i=0,j=0}^{\infty} (c_i^{(1)} c_j^{(2)}) u(t - (a_i^{(1)} + a_j^{(2)}))$$

The case of an n-fold convolution follows easily by induction. □

Using this lemma, f_{X_n} can be written as

$$\begin{aligned} f_{X_n}(t) &= \sum_{i_1,\ldots,i_n \in \mathbb{N}_0} (\underbrace{c_{i_1}^{(1)} \cdots c_{i_n}^{(n)}}_{=\hat{c}}) u(t - (\underbrace{a_{i_1}^{(1)} + \cdots + a_{i_n}^{(n)}}_{=\hat{a}})) \\ &= \sum_{m=0}^{\infty} \hat{c}_m u(t - \hat{a}_m) \end{aligned}$$

where m is the number of recovery steps taken and

$$\hat{c}_m = \sum_{\substack{m=i_1+\cdots+i_n \\ i_1,\ldots,i_n \in \mathbb{N}_0}} c_{i_1} \cdots c_{i_n}, \tag{6.1}$$

$$\hat{a}_m = n(\frac{t_\mathrm{S}}{n} + t_\mathrm{C}) + m(t_\mathrm{R} + \frac{t_\mathrm{S}}{n} + t_\mathrm{C}).$$

There are only a finite number of cases to write a natural number m as a sum of other natural numbers and therefore Equation (6.1) for $\hat{c}_m$ is well formed. To give a closed form for $\hat{c}_m$, the structural difference between c_0 and $c_i, i > 0$ must be considered (this reflects the fact that is does make a difference whether a normal step or a recovery step fails). The formula for $\hat{c}_m$ can be simplified by distinguishing two cases. Obviously, $\hat{c}_0 = \rho_1^n$. For $m > 0$, $\hat{c}_m$ is expressible as a sum over the number k of how many of the indices in the general sum are 0, and Equation (6.1) can be refined as follows:

$$\begin{aligned} \hat{c}_m &= \sum_{\substack{m=i_1+\cdots+i_n \\ i_1,\ldots,i_n \in \mathbb{N}_0}} c_{i_1} \cdots c_{i_n} \\ &= \sum_{k=1}^{n} \binom{n}{k} c_0^{n-k} \sum_{\substack{m=i_1+\cdots+i_k \\ i_1,\ldots,i_k \in \mathbb{N}}} c_{i_1} \cdots c_{i_k} \\ &= \sum_{k=1}^{n} \binom{n}{k} c_0^{n-k}\, \mathrm{psum}(m,k)(\, (1-\rho_1)^k\, (1-\rho_2)^{m-k}\, \rho_2^k\,) \end{aligned} \tag{6.2}$$

where psum is defined as follows:

Definition 6.2 ($\mathrm{psum}()$). *$\mathrm{psum}(m,k)$ is the number of possibilities to write m as a sum of exactly k* positive *natural numbers (were permutations of the sum are considered as different possibilities).*[3]

To simplify psum, the function nnsum is used:

Definition 6.3 ($\mathrm{nnsum}()$). *$\mathrm{nnsum}(m,k)$ is the number of possibilities to write m as a sum of k* nonnegative *natural numbers.*

Lemma 6.2 ($\mathrm{psum}()$ and $\mathrm{nnsum}()$). *For $m, k > 0$*

$$\mathrm{nnsum}(m,k) = \binom{m+k-1}{m}$$

$$\mathrm{psum}(m,k) = \begin{cases} 0 & \text{for } m < k \\ \mathrm{nnsum}(m-k,k) & \text{else} \end{cases}$$

[3] For example: $\mathrm{psum}(4,3) = 3$ since $4 = 1+1+2 = 1+2+1 = 2+1+1$.

Proof. The equation for nnsum can be shown with a simple induction on k: Write function $\mathrm{nnsum}(m, k+1)$ as $\sum_{i=0}^{m} \mathrm{nnsum}(m-i, k)$ (the new summand $k+1$ can be any value between 0 and m), use the induction hypothesis, apply $\binom{a}{b} + \binom{a}{b+1} = \binom{a+1}{b+1}$ iteratively from the right, and note for the last summand that $\binom{k-1}{0} = \binom{k}{0}$. $\mathrm{psum}(m, k) = \mathrm{nnsum}(m-k, k)$ is true because each of the k numbers must at least be equal to 1. □

Therefore, Equation (6.2) can be rewritten as

$$\begin{aligned}\hat{c}_m &= \sum_{k=1}^{\min(n,m)} \binom{n}{k} c_0^{n-k} \binom{m-1}{m-k} (\,(1-\rho_1)^k\,(1-\rho_2)^{m-k}\,\rho_2^k\,) \\ &= \sum_{k=1}^{\min(n,m)} \binom{n}{k} (\rho_1)^{n-k} \binom{m-1}{m-k} (\,(1-\rho_1)^k\,(1-\rho_2)^{m-k}\,\rho_2^k\,)\end{aligned}$$

for $m > 0$.

So far, $p_{\mathrm{cov}} = 1$ has been assumed. For imperfect detection coverage, the fault detection must detect all faults to be able to assume that a result is correct, i.e., it has to work for all m recovery steps that would occur in the case of $p_{\mathrm{cov}} = 1$. For m recovery steps, the probability of this case is equal to p_{cov}^m. The coefficient $\hat{c}_m$ for m recovery steps is hence modified to

$$\tilde{c}_m \stackrel{\mathrm{def}}{=} p_{\mathrm{cov}}^m \hat{c}_m.$$

Finally, given a deadline d at which the service has to be completed, the probability of successful completion under worst-case runtime assumptions is $P(X_n \leq d)$ for any given n. By the derivation of f_{X_n} and the above modification for p_{cov},

$$P(X_n \leq d) = \sum_{m=0}^{\infty} \tilde{c}_m H(d - \hat{a}_m) \tag{6.3}$$

with $\tilde{c}_m$ and $\hat{a}_m$ as defined above. Equation (6.3) is actually a finite sum, since only executions with a total execution time $\hat{a}_m$ smaller than d contribute to the probability. Therefore,

$$\begin{aligned}\hat{a}_{m_o} = n(\frac{t_{\mathrm{S}}}{n} + t_{\mathrm{C}}) + m_0(t_{\mathrm{R}} + \frac{t_{\mathrm{S}}}{n} + t_{\mathrm{C}}) \leq d \Rightarrow \\ m_0 = \left\lfloor \frac{d - t_{\mathrm{S}} - n t_{\mathrm{C}}}{t_{\mathrm{R}} + \frac{t_{\mathrm{S}}}{n} + t_{\mathrm{C}}} \right\rfloor\end{aligned} \tag{6.4}$$

is an upper limit for the number of possible recovery steps for deadline d ($m_0 < 0$ for $d < n(\frac{t_{\mathrm{S}}}{n} + t_{\mathrm{C}})$, which corresponds to the impossibility of meeting this deadline). This last observation completes the proof of the main theorem:

Theorem 6.1 (Responsiveness of a checkpointed service with fixed t_{S}). *For a service with a fixed execution time, the probability of meeting its deadline when n checkpoints are used is*

$$P(X_n \leq d) = \sum_{m=0}^{m_0} \tilde{c}_m \tag{6.5}$$

A closed-form solution for this Equation (6.5) would allow to determine the optimal n analytically, but such a form is not easy to find. A numerical solution, on the other hand, is quite simple. For such a numerical solution, it is important to note that there are only a finite number of checkpoints n that have to considered: too many checkpoints would imply that even in the fault-free case the deadline d cannot be met if $d < t_{\mathrm{S}} + n t_{\mathrm{C}}$ (by Equation (6.4)). Therefore, it suffices to compute Equation (6.5) only for such n with

$$1 \leq n \leq \left\lfloor \frac{d - t_{\mathrm{S}}}{t_{\mathrm{C}}} \right\rfloor$$

and choose the n that maximizes Equation (6.5); ties can be broken in favor of a smaller n.

6.4.2 Services with probabilistic execution time

In the analysis in Section 6.4.1, a fixed service execution time t_S has been assumed. Such an assumption is often not justified, but service execution times can often be modeled with a random variable (similar to the approach in Chapter 5). In this section, the random variable for this service execution time will be called I and it is assumed to be arithmetic;[4] and $F_I(t) = \sum_{i \in \mathbb{N}} p_i H(t - o_i)$ (i.e., the service needs o_i time units to complete with probability p_i, $\sum_{i \in \mathbb{N}} p_i = 1$, $p_i \geq 0$). The model adopted here is that the service execution time is random, but fixed once the service starts to execute; in particular, it does not change when the service has to be restarted from a previous checkpoint.

For such a service with probabilistically described execution time, it is no longer possible to reason about the number n of checkpoints; rather, some arbitrary checkpointing interval t_N has to be considered. The problem is then to find a t_N that maximizes the responsiveness of such a service at a given deadline d.

Let the random variable $\tilde{X}_{t_N}$ denote the time to successful completion of such a probabilistically described service when a checkpointing interval of length t_N is used. Computing the random distribution can be done using the techniques in the previous section. First note that

$$\Pr(\tilde{X}_{t_N} \leq t) = \sum_{i \in \mathbb{N}} p_i \Pr(\tilde{X}_{t_N}^{t_S = o_i} \leq t), \tag{6.6}$$

where $\tilde{X}_{t_N}^{t_S = o_i}$ is the random variable representing the time to successful completion of the service in the case that the service completion time equals o_i.

Since in general, t_N does not divide o_i, this is not exactly the same problem as in the previous section. It is possible, however, to divide this into two subproblems such that each matches the case of fixed execution times. The first subproblem corresponds to executing the service with blocks of length t_N. This execution continues as long as the remaining execution time is longer than t_N, and there will be $n' = \left\lfloor \frac{o_i}{t_N} \right\rfloor$ of these blocks. This subproblem is equivalent to the problem of Section 6.4.1 for a service with execution time $t'_S = \left\lfloor \frac{o_i}{t_N} \right\rfloor t_N$ and n' blocks. The runtime distribution can hence be calculated from Equation (6.5), with t'_S and n' substituted for t_S and n. The second subproblem corresponds to executing the remaining $t''_S = o_i - \left\lfloor \frac{o_i}{t_N} \right\rfloor t_N$ execution time of the service. And since this time is smaller then the checkpointing interval t_N, it will be executed in a single block, hence $n'' = 1$ and the runtime distribution can again be computed following Equation (6.5).

As an example, consider the case $o_i = 8$, $t_N = 3$, $t_C = 1$ in Figure 6.2. The first two blocks correspond to checkpointing a service with $t'_S = \left\lfloor \frac{o_i}{t_N} \right\rfloor t_N = \lfloor \frac{8}{3} \rfloor 3 = 6$ and $n' = \lfloor \frac{8}{3} \rfloor = 2$; the last block is equivalent to a service with $t''_S = o_i - \left\lfloor \frac{o_i}{t_N} \right\rfloor t_N = 8 - \frac{8}{3}3 = 2$ and $n'' = 1$.

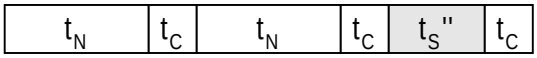

Figure 6.2: Fault-free execution with $o_i = 8$, $t_N = 3$, $t_C = 1$, resulting in $t'_S = 6$ and $t''_S = 2$ (shaded block).

Hence, the distribution of $\tilde{X}_{t_N}^{t_S = o_i}$ can be calculated as the convolution of the distribution of two auxiliary random variables, each calculated individually based on Equation (6.5): X' with t'_S and n', and X'' with t''_S and n''.[5] This results in the random distribution of $\tilde{X}_{t_N}$ for an arbitrary t_N and hence the responsiveness for a given deadline can be computed. Since for practical purposes, t_N is actually a discrete quantity (clock alarms can usually only be set to occur with a certain granularity, often 10 ms or 20 ms), enumerating all candidate

[4] The changes for non-arithmetic random variables are straightforward; replace the sum in Equation (6.6) with an appropriate integral.

[5] Using deadline d in the computation of X'' is an overestimation (it could be reduced to $d'' = d - n'(t_N + t_C)$) but nevertheless correct.

values is possible yet computationally expensive. It is possible to limit this overhead by bounding t_N, e.g., by the deadline.

6.5 Evaluations of theoretical model

In this section, it is attempted to give a feeling for the relative importance of the various parameters for the responsiveness of a service, using the results developed in Section 6.4 (further examples can be found in [138, 139]). The parameter space is seven-dimensional: t_S, t_C, t_R, λ, n, d and p_{cov}. As is commonly the case with a large number of parameters, it is quite difficult to give a complete overview of the behavior. Therefore, only a few selected figures will be presented.

An important motivation for using responsiveness as the optimization metric in the analysis is dealing with real-time services that have a deadline. In particular, optimizing the checkpointing interval so as to minimize the mean execution time is a poor choice for real-time services. For the runtime distribution shown in Figure 6.3 (with parameters $t_S = 10$, $t_C = 2$, $t_R = 1$, $\lambda = 0.1$, $p_{cov} = 1$), $n = 3$ minimizes the mean execution time (to 30.8). While $n = 5$ has a mean of 33.6, at a deadline of, e.g., $d = 70$, $P(X_3 \leq 70) = 0.984290$ and $P(X_5 \leq 70) = 0.995138$, which is over one percent point better. To obtain a given responsiveness, the deadline must be much longer with $n = 3$ than with $n = 5$: Ensuring a responsiveness of 0.99999 requires a deadline of 142.67 for $n = 3$, while with $n = 5$ this responsiveness is already achieved at $d = 115$.

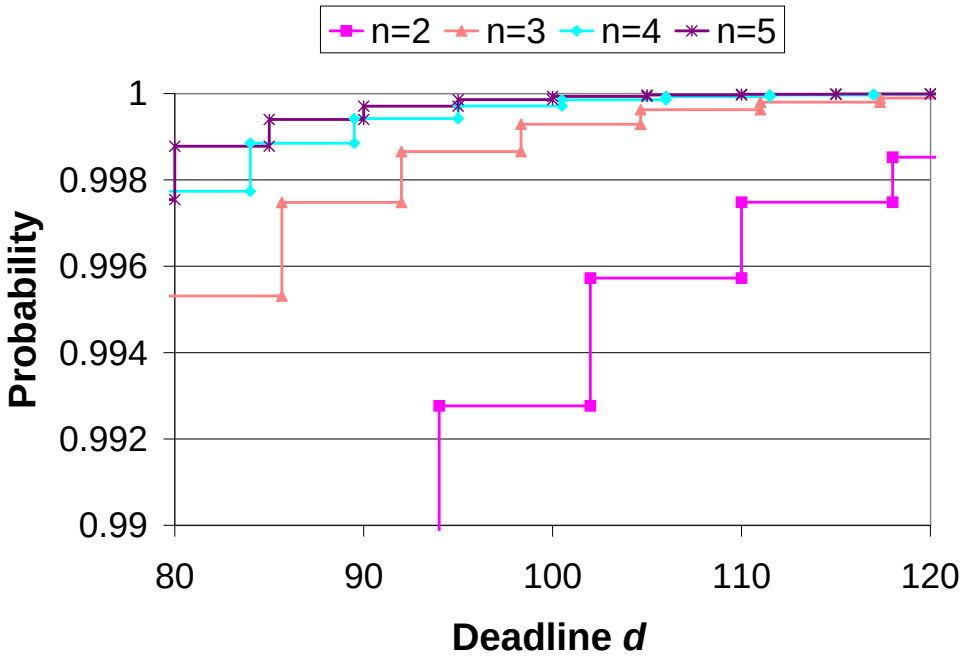

Figure 6.3: Completion time distributions $\Pr(X_n \leq d)$ shown over deadline d for various numbers of checkpoints n. Other parameters: $t_S = 10$, $t_C = 2$, $t_R = 1$, $\lambda = 0.1$, $p_{cov} = 1$.

Figure 6.4 shows the number of checkpoints n that maximizes responsiveness for increasing deadlines and three different service times. Note the characteristic shape of these curves: As the deadline increases, the best possible n increases in a roughly sawtooth-like pattern. An increase occurs when it is possible to execute an additional recovery step (before the deadline would expire). This is possible first for larger n, but if two different n permit the same number of recovery steps for a given deadline, the smaller n is preferable since it incurs smaller overhead and therefore a smaller chance of being hit by a fault—resulting in this characteristic sawtooth shape for small n. For large n, this effect is less pronounced since the difference in block length is sufficiently small to smooth out these bumps (when n starts growing rapidly, the responsiveness is already

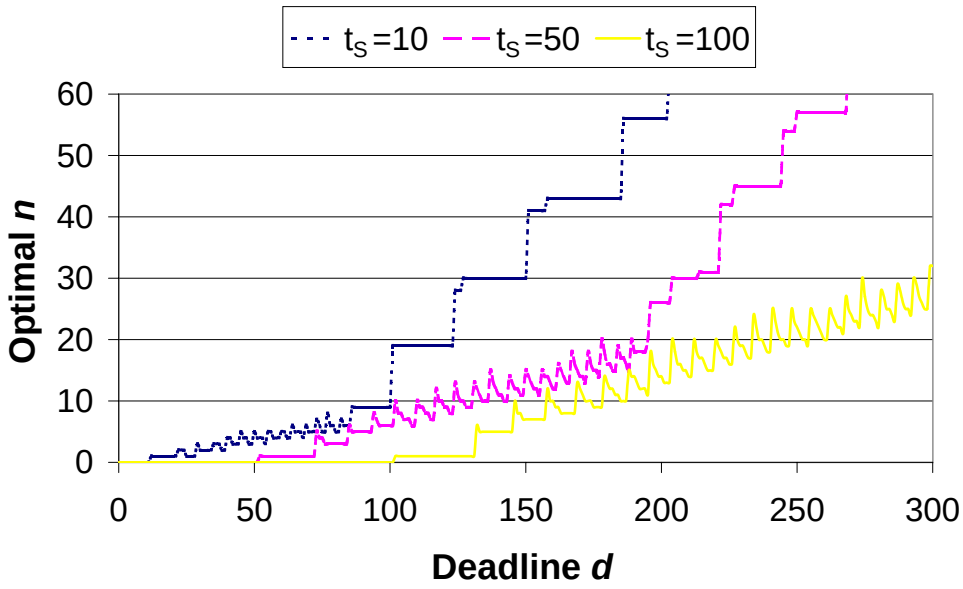

Figure 6.4: Number of checkpoints n maximizing responsiveness shown over deadline d for $t_S = 10$, $t_S = 50$, $t_S = 100$. Other parameters: $t_C = 2$, $t_R = 1$, $\lambda = 0.01$, $p_{cov} = 1$.

very close to 1). A simple rule of thumb is therefore to take the smallest n that allows the maximum number of recovery steps to be taken before the deadline would expire.

The impact of the coverage probability is shown in Figure 6.5. Here, p_{cov} is set to 0.6. A low coverage probability implies a large risk of not detecting a fault during the acceptance test and terminating with an invalid result. Therefore, there are two antagonistic tendencies: A larger number of checkpoints is preferable to avoid losing much work when recovery becomes necessary, a smaller number is better to reduce the chances of not detecting an error that is actually present (owing to the imperfect fault detection).

This tradeoff is further complicated by the fact that an acceptance test also influences the checkpointing time. Namely, to improve an acceptance test's coverage probability, it has to execute more tests and becomes more complicated and longer. Hence there is yet another tradeoff for the length of the checkpointing interval. Figure 6.6 shows the optimal number of checkpoints for a few different combinations of checkpointing time and coverage probability (an acceptance test can often take up the majority of the time of writing a checkpoint).

The examples so far have used carefully selected, but perhaps slightly unrealistic parameter values to highlight some salient points of the checkpointing analysis. Varying the mean lifetime and the service execution time t_S, the following Tables 6.1 and 6.2 on page 86 show the optimal n and the corresponding responsiveness for a somewhat more realistic example: Consider a large, long-running program that has a considerable amount of state, so that checkpointing takes $t_C = 30$ s and a recovery step takes $t_R = 10$ s. The acceptance test is the imaginary program is non-trivial, so that $p_{cov} = 0.999$. To be able to recover from faults, the deadline has to allow some slack, and consequently the deadline in this example is assumed to be fifty percent longer than the service execution time: $d = 1.5t_S$. Most importantly, service execution times of one hour, ten hours, and hundred hours are considered, combined with mean lifetimes of ten, hundred, thousand, and ten thousand hours. The results are as expected: with decreasing fault rate (increasing mean lifetime), the responsiveness becomes better and the optimal choice for n decreases.

The behavior with probabilistically described services as discussed in Section 6.4.2 is in principle similar, yet more complicated in detail. As an example, checkpointing with the same parameters as in Figure 6.4 was considered, however, the service execution time is here one of $10, 11, \ldots, 19$ time units, each occurring with 10% probability. Figure 6.7 shows the responsiveness for this service with three different deadlines ($d = 40$, $d = 50$, $d = 60$) when the checkpointing interval t_N is varied.

The optimal checkpointing interval when the deadline is varied is shown in Figure 6.8 (for the same

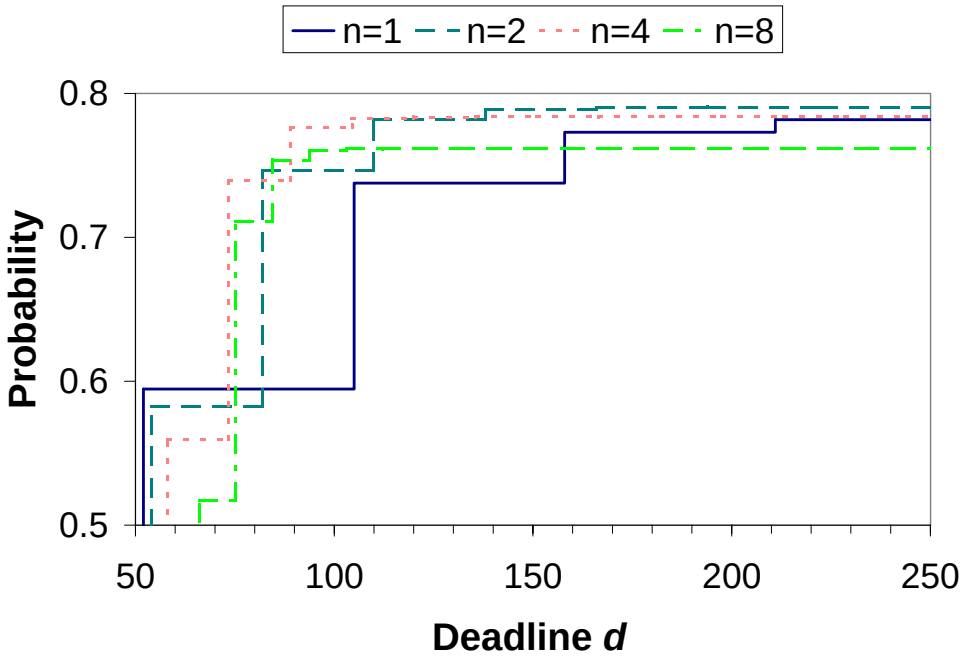

Figure 6.5: Completion time distribution $\Pr(X_n \leq d)$ shown over deadline d for various numbers of checkpoints n with coverage probability $p_{\text{cov}} = 0.6$. Other parameters: $t_{\text{S}} = 50$, $t_{\text{C}} = 2$, $t_{\text{R}} = 1$, $\lambda = 0.01$.

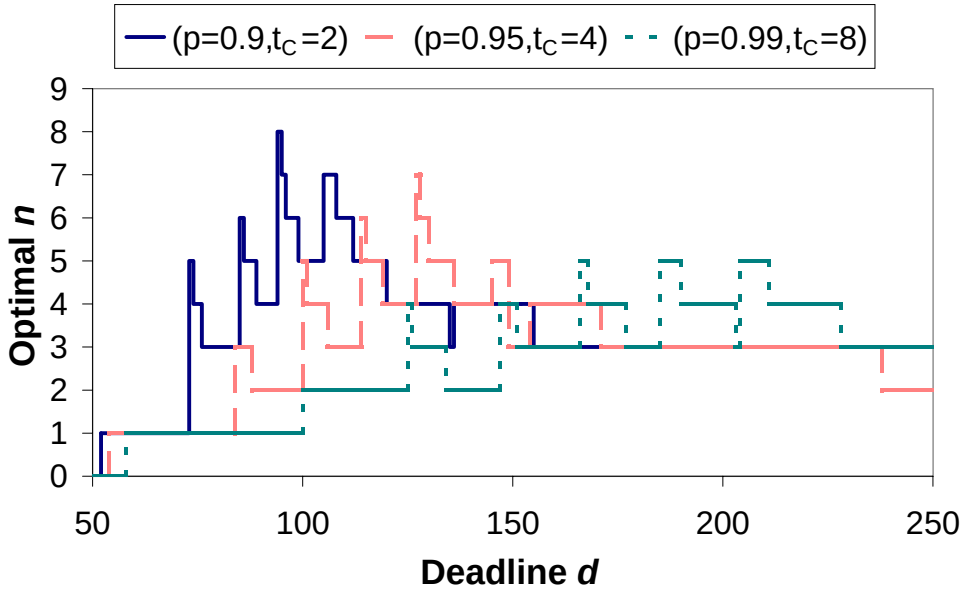

Figure 6.6: Number of checkpoints n maximizing responsiveness shown over deadline d for different $(p_{\text{cov}},t_{\text{C}})$ combinations. Other parameters: $t_{\text{S}} = 50$, $t_{\text{R}} = 1$, $\lambda = 0.01$.

	mean lifetime $1/\lambda$ in s			
t_S in s	36000 (10 hr)	360,000 (100 hr)	3,600,000 (1,000 hr)	36,000,000 (10,000 hr)
3600 (1 hr)	11	8	5	5
36000 (10 hr)	25	11	7	5
360000 (100 hr)	248	78	25	9

Table 6.1: Optimal number of checkpoints for varying mean lifetime $1/\lambda$ and service time t_S; other parameters: $t_C = 30$ s, $t_R = 10$ s, $d = 1.5t_S$, $p_{cov} = 0.999$.

	mean lifetime $1/\lambda$ in s			
t_S in s	36000 (10 hr)	360,000 (100 hr)	3,600,000 (1,000 hr)	36,000,000 (10,000 hr)
3600 (1 hr)	0.999890	0.999989	0.999999	> 0.999999
36000 (10 hr)	0.998958	0.999899	0.999990	0.999999
360000 (100 hr)	0.989632	0.998987	0.999900	0.999990

Table 6.2: Responsiveness (corresponding to optimal number of checkpoints shown in Table 6.1) for varying mean lifetime $1/\lambda$ and service time t_S, other parameters: $t_C = 30$ s, $t_R = 10$ s, $d = 1.5t_S$, $p_{cov} = 0.999$.

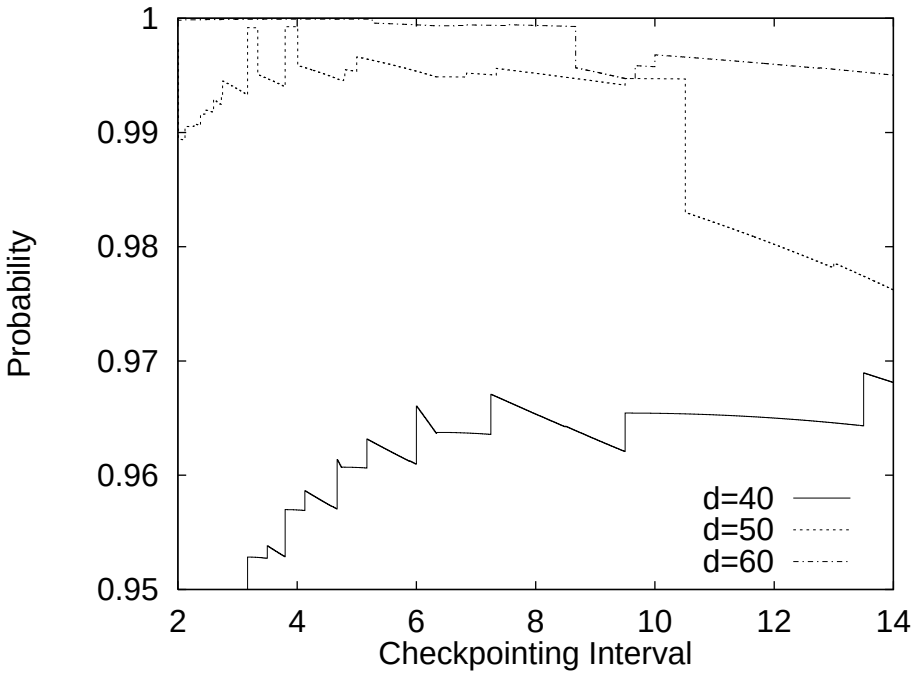

Figure 6.7: Responsiveness shown over checkpointing interval t_N for three different deadlines d. Other parameters: t_S is one of $10, 11, \ldots, 19$ with equal probability, $t_C = 2$, $t_R = 1$, $\lambda = 0.01$, $p_{cov} = 1$.

parameters). This behavior is in principle similar to the case of fixed execution time (Figure 6.4) since with increasing deadline, the optimal checkpointing interval becomes shorter (as more overhead is acceptable) and the responsiveness increases.[6] And again similar to Figure 6.4, Figure 6.8 shows a characteristic sawtooth pattern for the optimal checkpointing interval. It appears somewhat blurred—compared to the case of fixed execution time—by the randomness in the service execution.

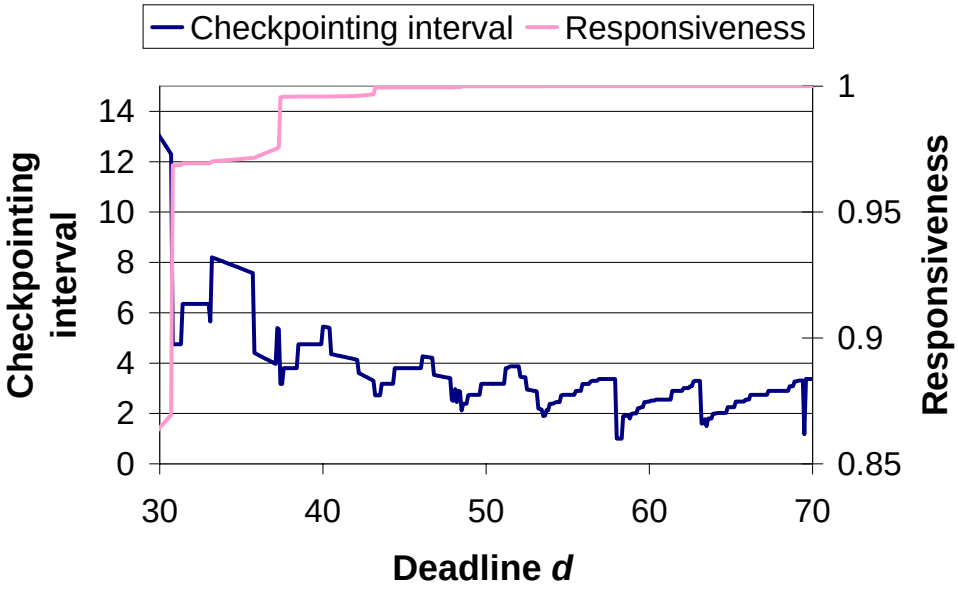

Figure 6.8: Optimal checkpointing interval and responsiveness shown over deadline d. Other parameters: t_S is one of $10, 11, \ldots, 19$ with equal probability, $t_C = 2$, $t_R = 1$, $\lambda = 0.01$, $p_{cov} = 1$.

6.6 Checkpointing the Calypso master

6.6.1 Implementation issues

Implementing checkpointing in the Calypso master process requires some modifications to the Calypso libraries as well as some additional information from the programmer. In particular, the programmer has to register an acceptance test function with the Calypso master that is invoked at checkpointing time, and an optional initialization function that is called during recovery (this function can, e.g., be used to open a new window on a graphical user interface). Additionally, the programmer can register supplemental data that should be checkpointed in addition to the state of the parallel execution. The checkpointing interval can be set directly with command line parameters to the Calypso master.

The state of the parallel execution is mainly a location marker, the value of local variables, the contents of the shared memory, the routine progress table, and the yet unapplied memory updates. These updates are problematic since their size can grow rapidly during a parallel step. And since the amount of data is an important parameter for the time it takes to write a checkpoint, growing state information is in conflict with the model assumption of bounded checkpointing time. Hence, only programs that have semantics compatible with immediately applying updates to the shared memory (see the discussion in Section 4.1) are suitable for this checkpointing approach.

For the worker processes, recovery of the master process is almost completely transparent. If the master performs a rollback, a worker detects a communication problem with its master, resets itself to an initial state and tries to recontact the master (a bounded number of times, in case the master has crashed permanently).

[6] Note that a larger n corresponds to a smaller t_N.

However, this implies that an unpredictable amount of work in the worker is lost and progress is delayed by the reconnection overhead. Both factors are not included in the analytical model.

In its current prototypical implementation, the checkpointing suffers from some limitations that somewhat restrict the way in which Calypso programs can be written (e.g., handling of stack variables). These restrictions can be overcome by using a standard commercial checkpointing library, e.g., [298].

6.6.2 Some experiments

The test program

To assess the performance and responsiveness impact of checkpointing and also judge the responsiveness with and without checkpointing, some experiments based on the Calypso example program as introduced in Section 4.4 were performed. The focus here is on granularity, fault rate (for the master process), and the number of the checkpoints as recommended by the analysis for a number of different deadlines.

Unlike the experiments in Section 4.4, which showed times for an individual parallel step, this section presents numbers for overall program execution: wall clock times from program start to successful or unsuccessful completion, including all overheads like starting remote workers. This is in accordance with an end-to-end concept of service execution.

More specifically, a program with twenty parallel steps was considered, where each step takes one second to execute on a single machine. To assess the impact of granularity, the length of a single routine was varied, and the number of routines per step was chosen accordingly to result in one second execution time per step. The objective of these experiments is to measure the runtime distribution of this program in different scenarios, e.g., with fault injection at different fault rates.

A few remarks on statistics

A few remarks regarding the statistical relevance of such experiments are in order. Repeating such an experiment (with the same parameter values) n times results in a sample $x_1, \ldots, x_n$ of values, here for the runtime of a program. This sample gives rise to an empirical distribution $S_n(x_i)$ that is an estimation of the true, in general unknown distribution of the underlying stochastic process. The basic justification for deriving any information out of these samples is the theorem of Glivenko and Cantelli: With probability 1, the supremum difference between the empirical and the true distribution vanishes if n goes to infinity (cp., e.g., HARTUNG et al. [103, p. 121]).

If the true distribution of the underlying random variable is known, it is often possible to estimate parameters (e.g., the mean) of this distribution. In the present case, however, no such knowledge about the true distribution is available—the empirical distributions of these experiments fail, e.g., a test for normality. Handling such problems requires methods of non-parametric statistics.

An adaption of the Kolmogorov-Smirnov goodness of fit test is suitable for this problem. Given a sample $\{x_1, \ldots, x_n\}$ of size n with the empirical distribution $S_n(x)$, some information about the true but unknown distribution $F(x)$ can be obtained with a confidence level $1 - \alpha$ (see HARTUNG et al. [103, p. 240] for details): The true distribution is bounded by two limiting functions L_{high} from above and L_{low} from below, where $L_{\text{high}}(x) = S_n(x) + \frac{1}{\sqrt{n}}\mathrm{d}_{n;1-\alpha}$ and $L_{\text{high}}(x) = S_n(x) - \frac{1}{\sqrt{n}}\mathrm{d}_{n;1-\alpha}$, respectively. Here $\mathrm{d}_{n;1-\alpha}$ are the critical values of the Kolmogorov-Smirnov test for confidence level $1 - \alpha$: $\mathrm{d}_{n;0.95} \approx 1.36$, $\mathrm{d}_{n;0.98} \approx 1.52$, $\mathrm{d}_{n;0.99} \approx 1.63$ for $n > 40$.

The two functions L_{low} and L_{high} together give a simple confidence band for an unknown distribution around an empirical distribution. The advantage of the Kolmogorov-Smirnov estimation is that the width of this band does not depend on the actual sample values and therefore allows a simple computation of the number of experiments to achieve a desired precision for the estimation of the true distribution. For example, for a 5% width of the confidence band at a confidence level of 95%, $n \geq (\frac{2\mathrm{d}_{n;0.95}}{0.05})^2 \approx (\frac{2.72}{0.05})^2 \approx 2960$ samples are needed.

This method of computing confidence bands is only a conservative estimation. If the actual sample values are also used, the confidence band can be tightened as follows. The estimation of $F(x)$ at any sample point

can be considered as an estimation problem for the success probability of a binomially distributed random variable. For this problem, confidence intervals can also be derived. For large n and not too extreme success probabilities, the binomial distribution can be approximated with the normal distribution—HARTUNG et al. [103, p. 203] discuss this problem in detail and also show methods to improve the approximation accuracy (the Pearson-Clopper statistics) if the normal approximation cannot be used. While this improves the confidence band over the Kolmogorov-Smirnov estimation, it does not allow any simple a priori estimation of the number of experiments necessary to achieve a certain width of the band. However, it allows to abort an experiment once enough samples have been collected so that the width of the confidence band is sufficiently small. For all the following experiments, a confidence level of 95% was chosen and the maximum acceptable width of the confidence band is set to 5%.

This quite evidently limits the number of possible experiments. A full set of experiments for even a few settings for the above mentioned parameters granularity, deadline, and fault rate—which are by no means all possible parameters imaginable—would take many months of CPU time and is impractical. Therefore, some preliminary experiments with only 100 runs each were performed for a number of different parameter combinations. Based on these experiments, parameters were selected that resulted in typical and good behavior—e.g., out of the granularities 1, 5, 10, 50, and 100 ms that were considered in the preliminary experiments, 50 ms was chosen since it behaved typically for both plain Calypso and Calypso with checkpointing (and also for the replicated Calypso discussed in Chapter 7).

The results

Running the test program (described at the beginning of this section) on four machines (Pentium 90, 10 Mbps Ethernet) resulted in a distribution of the total runtime as shown in Figure 6.9, each curve showing the distribution for a different granularity. A remarkable feature of these distributions is the presence of a number of plateaus where the distribution does hardly change. These plateaus are due to the inclusion of the startup procedure of the program, in particular starting the worker processes on remote machines.[7] First, the master program starts and immediately forks off four `rsh` processes that in turn start the remote worker processes. Some of these worker processes might start right away, some might be delayed. However, the program starts executing even when only some workers are present, running slowly at the beginning and increasing in speed when all the workers have joined. While it could be debatable to include this overhead in the measurements, it does reflect the actual behavior of a real program much better than artificially cutting out only some phases of a program and neglecting startup costs.

Apart from this plateau characteristic, the results are rather straightforward: small granularity results in higher overheads and longer runtimes. In particular, 50 ms granularity shows a typical behavior, compromising between low overhead and reasonable load balancing, and will therefore be used in all the following experiments (100 ms granularity has a slightly smaller mean execution time of 9.06 s as opposed to 9.09 s for 50 ms granularity, but also a slightly larger variation coefficient; other granularities result in notably slower execution). Figure 6.9 gives an overview of the runtime distributions for this problem, without checkpointing or fault injection, for a number of different granularities g.

Figure 6.10 shows the runtime distribution for this granularity for over 2000 runs, resulting in a confidence band width of 4.17 % (at a confidence level of 95%) according to the binomial-based estimator (the middle curve shows the empirical distribution, the two outer curves the lower and upper boundary of the confidence band). It is interesting to note that for such a large number of experiments, the step-wise characteristic due to worker startup is blurred, but still visible to a certain extent.

To assess the impact of checkpointing on the runtime distribution of this program, a number of fault injection experiments were conducted. The fault rate in these experiments was unrealistically high and is merely to validate the checkpointing implementation and should not be misconstrued as an indication of the actual dependability of the Calypso system. For the fault injection, exponential reliability functions with mean lifetime 20 s and 50 s were considered (10 s mean lifetime has also been investigated in the preliminary experiments, but results are not reported here). Figure 6.11 shows the runtime distribution for the test program

[7] These plateaus do not appear if only the execution time of repeated parallel steps is measured.

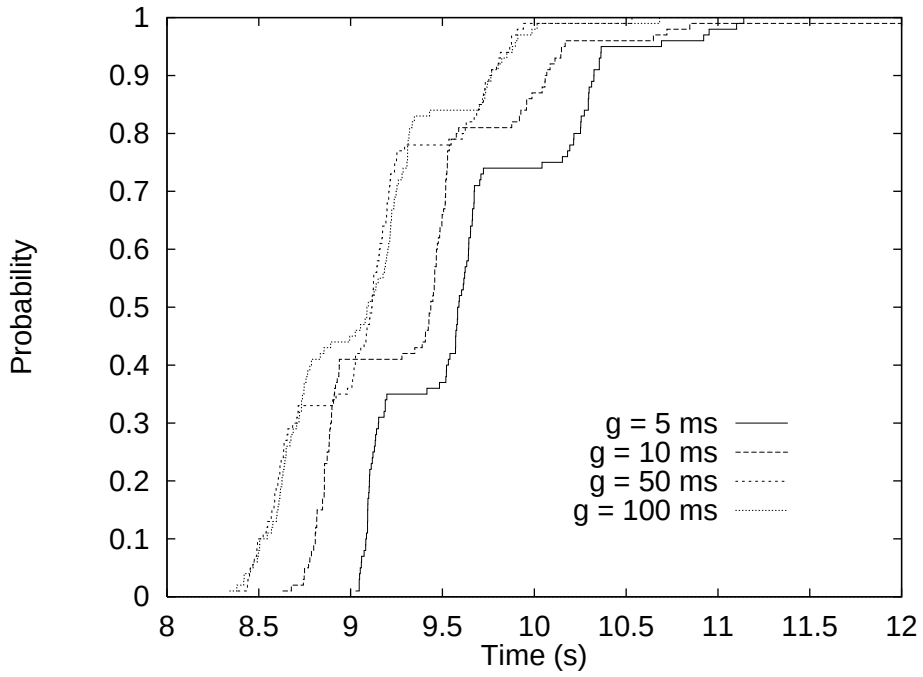

Figure 6.9: Runtime distribution of a complete Calypso program with different granularities g, no checkpointing or fault injection, 100 runs each.

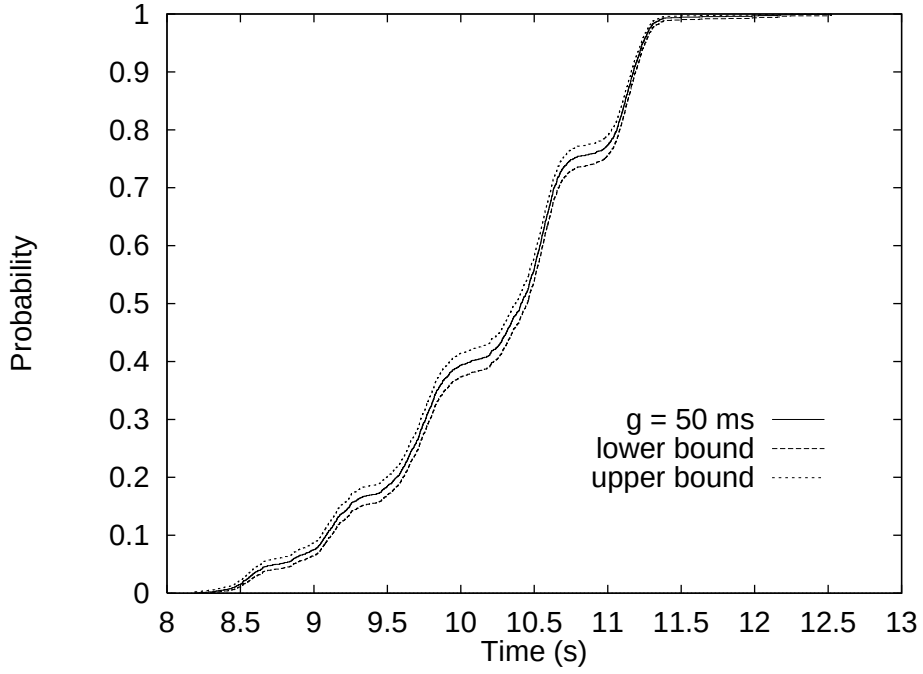

Figure 6.10: Runtime distribution of a complete Calypso program with granularity 50 ms and upper and lower bounds of the confidence band, confidence band narrower than 5%.

with 50 ms granularity and faults injected according to these two fault rates; the confidence band for both cases is narrower than 5%.

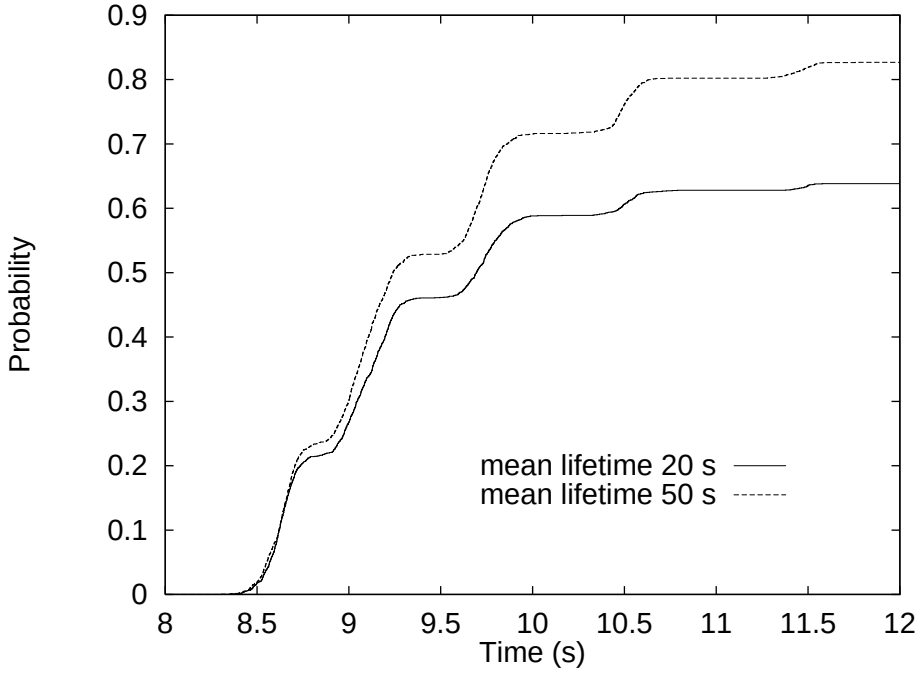

Figure 6.11: Runtime distribution of a complete Calypso program with fault injection for mean lifetime of master 20 s and 50 s, no checkpointing, granularity 50 ms, confidence band narrower than 5%.

To apply the checkpointing analysis presented above, a deadline has to be selected to compute an optimal checkpointing interval. Since the service itself takes 5 s to compute, plus about 6 s startup time in the worst case, a reasonable deadline is 16 s: This deadline gives enough leeway for one redundant service execution even after all the startup overhead has been executed (in the preliminary experiments, also 11 s and 21 s were considered as deadlines, combined with other granularities as well). Using a fixed service execution time for the parallel step is a simplification, however, this can be justified as no worker faults are considered. Even with worker faults, the examples from Section 5.5.2 indicate that for reasonably small fault rates and fixed routine runtimes the density of eager scheduling's runtime has one dominating peek that can be used to approximate a fixed runtime.

Based on service parameters[8] and the fault rates mentioned above, the analysis yields $n = 7$ as best number of checkpoints for both 20 s and 50 s mean time between failures.[9] Due to the small overhead caused by checkpointing, the overhead for the entire program in the fault-free case is almost negligible.

To check these analytical results, a Calypso program enhanced with checkpointing functionality was subjected to faults which were distributed according to a Poisson process with the corresponding mean: After a randomly selected time, a flag is set to indicate that a fault has occured. Every time a checkpoint is written, this flag is tested and, if set, the Calypso master process terminates itself. It is then restarted by a wrapper process, and a new failure time is again selected randomly. Owing to this restart, the mean lifetime of a single process is also its Mean Time Between Failures (MTBF).

Figures 6.12 and 6.13 show the runtime distributions resulting from these experiments for a number of

[8] $t_{\mathrm{C}} = 2$ ms, $t_{\mathrm{R}} = 70$ ms, $p_{\mathrm{cov}} = 1$ in this environment. Unfortunately, the Linux version used for these experiments allows only to flush output buffers, but it is not possible to set stream parameters such that the flush function call only returns after all data has actually been written to a disk. This would be desirable for checkpointing and would also increase the checkpointing overhead t_{C}; numbers can be found, e.g., in [278].

[9] More precisely, $n = 7$ is the smallest n such that the analysis predicts a success probability larger than $1 - 10^{-12}$.

different values of n ($n = 1$ and $n \geq 10$ show inferior performance). As the checkpointing overhead in these experiments is very small, the curves for the various values of n show different behavior only after about 11 s: The behavior for runtimes smaller than 11 s corresponds to the fault-free case (see Figure 6.10), and the impact of the various checkpointing intervals manifests itself only in the various lengths of the recovery block, which become relevant only for longer execution times. Since the actual responsiveness values are difficult is see in these graphs, Table 6.3 shows the responsiveness for MTBF 20 s at a deadline of 16 s, Table 6.4 for MTBF 50 s (confidence intervals are based on the Pearson-Clopper statistics).[10]

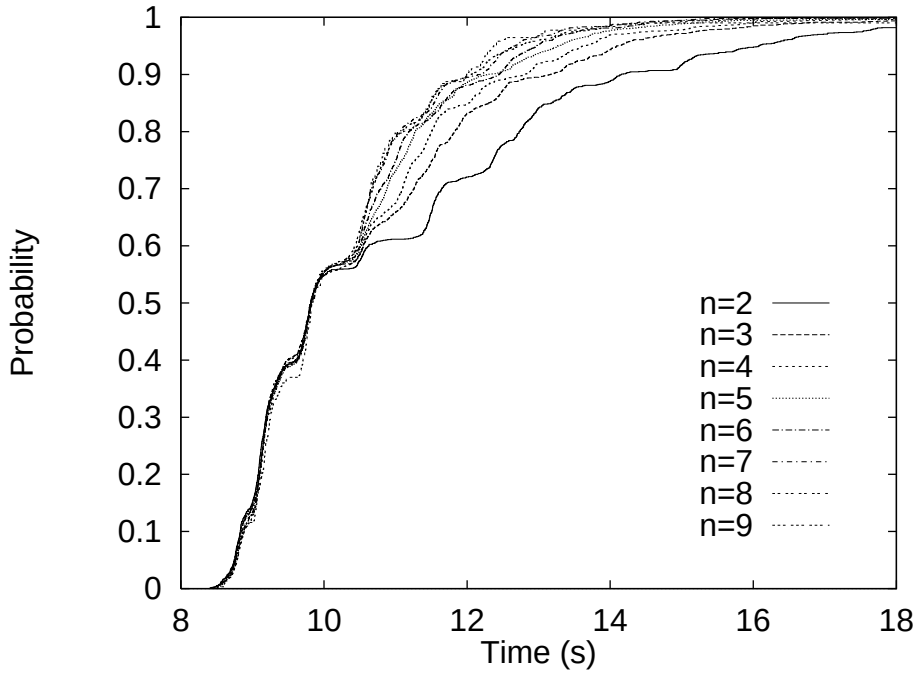

Figure 6.12: Runtime distribution of a complete Calypso program with checkpointing enabled, fault injection with MTBF 20 s, granularity 50 ms, confidence band narrower than 5 %.

Table 6.3 and 6.4 allow two main conclusions. The first is that the analytical value of $n = 7$ is indeed a good approximation—for both fault rates, it matches an optimal value of n. However, the responsiveness values obtained for different n are not significantly different in a stochastic sense.[11] Since the analysis does make some simplifying assumptions, some differences between analysis and theory in a real application are not very surprising. For example, the actual responsiveness values are lower than analytically predicted, a fact that can be attributed to the reconnection delay and work lost by the worker process when the master process is restarted. The second observation is that Tables 6.3 and 6.4 indicate similar characteristics as the analytical results for a (much simpler) service with a randomly distributed execution time from Figures 6.8: there is a certain range of values for n where the impact on the responsiveness is only small. These two observations together can justify the use of the analytical results for a service with a fixed runtime. The main advantage is that it is not necessary to determine the probability distribution of a service's execution time to use checkpointing for responsiveness, but that the simpler analysis for fixed execution times results in acceptable approximations.

[10] For a mean lifetime of 200 s (results not shown), a deadline of 16 s is long enough to ensure that all experiments finish before this deadline as long as $n > 1$. Indeed, for $n > 5$, $d = 14$ s is practically almost met, and the difference in success probabilities is small.

[11] In more detail: a χ^2 test [243, p. 448] (computed with the SAS statistics program) for the responsiveness of the various checkpointing numbers and fault rates yields the following results: For MTBF 50 s, $n = 1, 2$ have significantly lower responsiveness with a deadline of 16 s, all other values for n do not result in significantly different responsiveness. For MTBF 20 s, $n = 7$ is significantly better than $n = 1, \ldots, 4$, but statistically not different from other values of n (all statements at a 95% confidence level).

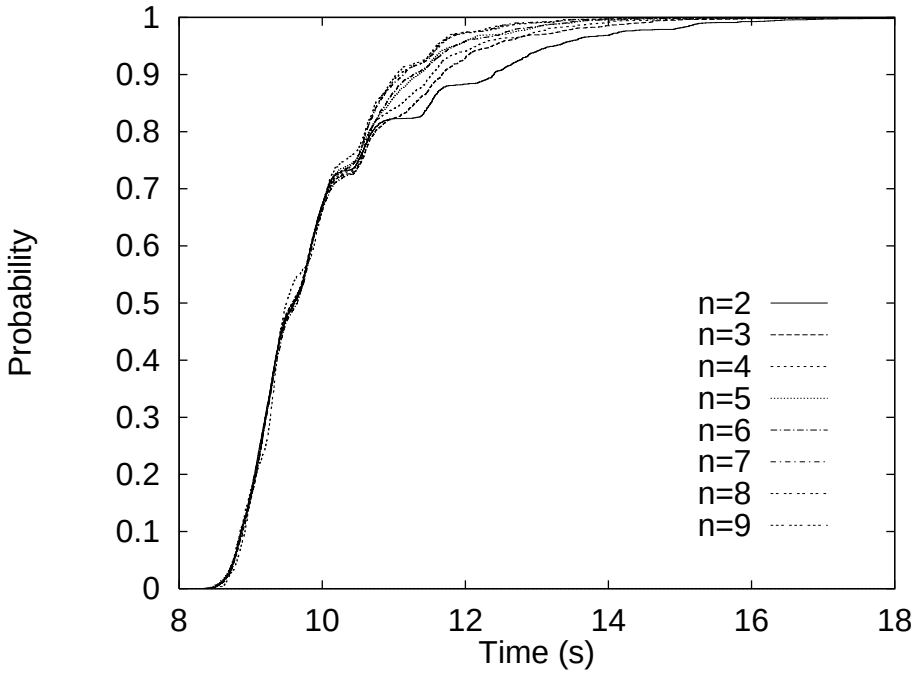

Figure 6.13: Runtime distribution of a complete Calypso program with checkpointing enabled, fault injection with MTBF 50 s, granularity 50 ms, confidence band narrower than 5 %.

n	value	low	high
1	0.8930	0.8452	0.9316
2	0.9475	0.9361	0.9574
3	0.9853	0.9785	0.9903
4	0.9904	0.9848	0.9943
5	0.9946	0.9902	0.9974
6	0.9968	0.9930	0.9988
7	0.9979	0.9945	0.9994
8	0.9978	0.9945	0.9994
9	0.9904	0.9722	0.9980
10	0.9904	0.9722	0.9980

Table 6.3: Responsiveness of Calypso program with varying number of checkpoints at deadline $d = 16$ s and MTBF 20 s, columns show value estimate and lower and higher end of 95% confidence interval.

n	value	low	high
1	0.9585	0.9227	0.9809
2	0.9931	0.9897	0.9956
3	0.9979	0.9957	0.9992
4	0.9983	0.9962	0.9994
5	0.9988	0.9970	0.9997
6	0.9988	0.9970	0.9997
7	0.9991	0.9974	0.9998
8	0.9991	0.9974	0.9998
9	0.9984	0.9953	0.9997
10	0.9984	0.9953	0.9997

Table 6.4: Responsiveness of Calypso program with varying number of checkpoints at deadline $d = 16$ s and MTBF 50 s, columns show value estimate and lower and higher end of 95% confidence interval.

6.7 Conclusions

In this chapter, the problem of using checkpointing for responsiveness has been considered. It has been shown that, while checkpointing is a well researched paradigm for fault tolerance, optimizing the responsiveness of a service requires different decisions than traditional optimization criteria like mean execution time.

One particularly important parameter of checkpointing is the number of checkpoints to take during service execution or, equivalently, the interval between writing checkpoints. Other parameters (e.g., time to write a checkpoint, fault rate) are commonly given. Therefore, it is an optimization problem to choose a checkpointing interval that maximizes the responsiveness of a service.

This optimization problem is solved by an analysis that makes realistic assumptions about checkpointing—e.g., that no acceptance check is perfect or that fault detection does not necessarily happen instantaneously. The analysis allows a simple and efficient numerical computation of the optimal checkpointing interval for services with both a fixed or a probabilistically described execution time.

The results of this analysis are then used to obtain checkpointing intervals for a Calypso version extended by checkpointing functionality. The effects of checkpointing in Calypso are evaluated with a number of experiments. These experiments indicate that checkpointing is indeed a suitable mechanism for increasing the responsiveness of a Calypso program, even under heavy fault injection. For reasonably large deadlines (on the order of 1.5 times the service execution time), the deadline is met with a very high probability. The experiments also show that the responsiveness of such a service is fairly robust against variation of the number of checkpoints as long as a value in the vicinity of an optimal value is used.

6.8 Possible extensions

There are a number of possibilities to extend the theoretical analysis. A practical issue is reducing the granularity of the fault detection by introducing a watchdog timer, which also extends the covered fault classes by directly including crash fault. Also, an extension to distributed checkpointing is conceivable.

The theoretical model shares a basic shortcoming with a lot of other research on checkpointing: the fault process is assumed to be Poisson. PLANK and ELWASIF [225] show in a number of experiments that the fault behavior of workstations follows a Poisson model only with vanishingly small probability; they do not attempt to characterize the actual (rather complicated) fault processes. Curiously enough, PLANK and ELWASIF also show that despite this mismatch in assumptions, the Poisson-based results for using checkpointing to optimize mean execution time are nonetheless an acceptable approximation. It would therefore be interesting to see if a

better solution for optimizing the responsiveness could be found with other fault models but such models are considerably harder to analyze owing to the interaction of faults across different checkpointing intervals.

With regard to Calypso, including the amount of work lost by the workers when recovery takes place and the workers' reconnection time during recovery can likely be mapped to the overall recovery time t_R. However, this requires some further investigations.

An interesting possibility for checkpointing in Calypso—and similar systems—appears when combined with a resource management system like the one described in Chapter 8. This system cyclically allocates time slices (e.g., 20 ms every 100 ms) to a parallel program. It is imaginable to allocate to the master process some additional runtime outside of its normal slice during which it could perform its checkpointing. The advantage is that from the workers' perspective, the master is not delayed by the checkpointing, it can perform its acceptance check, and even initiate a rollback recovery while the worker processes are kept dormant.

Chapter 7

Replication for Responsiveness

In this chapter, the use of replication for increasing responsiveness is considered. A wrapper-based approach to replication is presented that intercepts a program's input/output (I/O) interactions and uses this information to implement a flexible replication layer. This layer uses the Totem group communication protocol to remove a single point of failure; the response time distribution of Totem is investigated with some experiments. Based on this layer, replication is used in Calypso to ameliorate the problem of its single point of failure.

7.1 Introduction

In the previous Chapter 6, the use of checkpointing, an example for redundancy in time, has been investigated as a fault-tolerance paradigm for responsiveness. A definite advantage of checkpointing is its good analyzability. A disadvantage is the fact that during a rollback, the application process, e.g., Calypso's master, is not able to process service requests and the progress of the program is stalled. This rollback time can become substantial if crash faults of the master machine itself (and not only the master process) are considered. Long service interruption caused by, e.g., rebooting can be unacceptable in many environments.

Since the use of a parallel system like Calypso demands a reasonable amount of resources, ample redundancy in space is available, allowing the disadvantage of checkpointing to be overcome by replicating the master process on multiple machines. As long as at least one replica is operational, the service is permanently accessible. Replication can, unlike checkpointing, also serve the additional purpose of spreading the load of a single master process to multiple machines, potentially removing performance bottlenecks.

One approach to use replication for both fault tolerance and load balancing would be to tightly couple these two functionalities in a single, special-purpose implementation of Calypso. However, such a special implementation would limit the generality of the proposed solutions. A closer look at a replicated Calypso master process shows that replication interacts with two more or less independent functionalities. The first one is the handling of sequential activities, namely input/output (in the Calypso model, I/O is only allowed during the sequential steps of a program). The second one is handling parallel steps, namely the interactions with the workers.

A trivial approach to replication would thus be to execute k copies of the master process and assign workers to a single master, and the masters interact only with their assigned workers and ignore the other workers. Doing so removes the need for dealing with the replication in the parallel step and only leaves open the question of dealing with the I/O problem. However, it divides the available processing capacity by k and is therefore unacceptable. A reasonable solution requires masters to share the results contributed by all workers. This sharing should be implemented with as little synchronization and data exchange among the masters as possible since they are mere overhead compared to the non-replicated case. The masters' input/output, on the other hand, must be carefully synchronized to guarantee a correct program semantics and to make the replication transparent to the environment (GUERRAOUI and SCHIPER [98] call this transparency requirement of behaving equivalently to a non-replicated version "linearizable").

Comparing input/output activities and parallel activities shows that input/output is usually a rather rare

event—the program would not have been parallelized in the first place were it not for the computationally long phases of the parallel step. It is therefore unacceptable to burden the replication of the parallel steps with the tight synchrony requirements of the input/output processing. It will turn out later that it is indeed possible to implement parallel steps on replicated masters with considerably simpler synchrony requirements than input/output.

Solving the problem of replicated I/O can thus be considered independently from dealing with the parallel steps and the context of Calypso. Ideally, a non-intrusive solution should be found that is applicable to many other applications as well. In Section 7.2, a wrapper-based solution for this problem is introduced, capable of handling legacy software that conforms to standard I/O behavior.

This wrapper uses the Totem group communication protocol to efficiently implement the necessary synchrony requirements. Unlike checkpointing, the interactions of such a service with its environment are more complicated (e.g., scheduling of processes on independent machines, the communication network, etc.). These interactions make a reasonably accurate mathematical treatment more difficult. Therefore, an experimentally oriented approach was chosen to investigate the impact Totem might have on the responsiveness of a system. Some results of these experiments are presented in Section 7.3 and are contrasted with an existing analysis of Totem.

On the basis of this solution for the I/O problem of replicated processes, it is then possible to concentrate on the Calypso-specific problems of replication. In Section 7.4, a design (along with some measurements) for a replicated Calypso master is described that works with considerably simpler synchrony requirements than those needed for I/O, validating the design decision to separate these two problems and provide independent solutions. Finally, conclusions for this chapter are presented in Section 7.5 and Section 7.6 contains a discussion of possible extensions.

7.2 A wrapper approach to replication

7.2.1 Introduction

As pointed out in Section 7.1, a non-intrusive solution for increasing the dependability of applications is needed. Dependability of applications is a much sought-after property in many application domains. In new applications, it is possible to integrate mechanisms for fault tolerance, such as replication, from scratch. This option is often not available for existing applications, where it might be impossible to modify the application because the source code is not available, or merely impractical because modifying existing, complex applications is difficult to do without introducing errors or severely hampering performance. The latter would be the case for the Calypso system if a straightforward replication approach were to be taken.

These difficulties motivate the need for a solution that provides replication-based fault tolerance even to legacy applications. Such a solution has to conform to standard forms of interactions, since it can only observe an application's behavior from outside, via the application's interfaces. In this sense, the desired solution should be interface-based.

Natural interfaces (at least in a UNIX environment) are the standard input/output streams of any process. Other interfaces might be file I/O, network connections, or remote object invocations as often found in object-oriented, distributed systems. Here the focus is on the standard input/output streams, as naturally used by any interactive application, e.g., Calypso programs. However, the approach described here does not prevent a generalization to other types of interfaces.

As an example scenario, consider a Calypso program running with two replicated masters on two separate machines. The user should be able to interact with the program even if either of the two machines has crashed and even if the user's interface (e.g., a terminal window) ran on a crashed machine. This requires the ability to process input and output in a distributed fashion and to provide location transparency to user interfaces.

The Fault-Tolerant Distributed I/O (FT-DIO) system, developed in joint work with A. Polze and M. Werner [136], fulfills all these requirements. It provides different versions of wrappers that filter input and output of legacy applications and supports consistent replication of the program as specified by a user. A wrapper in this context is a small program that starts the actual, controlled program as a child process and intercepts the

input/output of its child. The replication is largely transparent both to the user and the program itself. The replicas of the application process usually act as hot standby for the application (since all processes process input and produce output), but cold and warm standby are also feasible with FT-DIO.

One key implementation technique of FT-DIO is an object hierarchy of fault handlers that reflects, by means of inheritance, the abstract hierarchy of fault classes. The inheritance relationships are organized so that it is possible (by means of polymorphism) to adapt—even at runtime—to changing fault models. This hierarchy also makes it easy for a programmer to create handlers for new fault models.

The flexibility of this handler hierarchy makes it possible to overcome another commonly found restriction of replication. A typical necessary condition for replication is consistency among the replicas: started in the same state, all replicas undergo the same sequence of state changes if they are presented with the same input, and consequently all produce the same output—if the results differ, then a fault is assumed. For most of the fault models discussed later, this is true as well. But it is also possible to use handlers that allow for different kinds of non-determinism among the replicas. A simple example would be randomized decision algorithms arriving at different solutions, where a positive answer takes precedence over a negative one; another example would be analytic redundancy [253] among heterogenous replicas.

Work related to FT-DIO is described in Section 7.2.2. The user interface of FT-DIO is shown in Section 7.2.3, and the correspondence between fault models and handler classes is briefly discussed in Section 7.2.4. Some implementation issues are discussed in Section 7.2.5 and measurements are shown in Section 7.2.6. Finally, the description of FT-DIO is concluded in Section 7.2.7 and in Section 7.2.8, some possibilities to extend this work are illustrated.

7.2.2 Related work

Providing fault tolerance transparently to an application has been actively pursued by approaches that customize hardware and/or operating system in some way. Examples include product lines from Stratus or Tandem; an overview of hardware-based approaches can be found in [260].

GUERRAOUI and SCHIPER [98] give an overview of software-based replication for fault tolerance. This paper only considers ways to handle crash faults and highlights primary-backup and active replication as main approaches to replication. FT-DIO in its most extended form is an example for active replication. Replication in general is a popular design and programming paradigm for fault tolerance; group communication protocols as described in Section 3.3.2 are typical examples.

A popular approach for dealing with legacy software is the Common Object Request Broker Architecture (CORBA) [216]. Such a middleware architecture suggests itself as a place to implement fault tolerance, e.g., by replicating server objects. The existing CORBA standard does not specify any fault-tolerance functionality, however, there is some activity towards it: A request for proposal “Fault-tolerant CORBA Using Entity Redundancy” [217] has been issued in April 1998. The results of this initiative are still open. The related problem of accessing a group of replicated server objects from a non-replicated client is addressed by client-access protocols [129].

Interestingly enough, two recently proposed systems (NCAPS and TFT) share some similarities with FT-DIO. LARANJEIRA describes NCAPS [162], a warm standby solution to achieve high availability for applications. Unlike FT-DIO, the application has to be aware of this mechanism and must be correspondingly modified (see an extended description of this system in Section 3.3.3).

The Transparent Fault Tolerance (TFT) approach presented by BRESSOUD [44] is very closely related to FT-DIO. TFT interposes a supervisor agent between the application and the operating system, emulating the operating system’s interface and distributing the system calls of a primary replica to a (in the current version single) backup replica. The backup’s supervisor replays all operating system activities to the backup copy, thus ensuring replica determinism between primary and backup. In case a failure of the primary process is detected, the backup process is promoted to primary status. While TFT handles more general interaction mechanisms of application and environment than FT-DIO, it has its own disadvantages: The application needs to be relinked with a modified system library; complete determinism between replicas might not even be desirable for diversified replicas (which are relatively easy to handle with FT-DIO); FT-DIO does not need

an explicit primary replica, which allows a faster response if one of the processes fails since no failover of any kind is necessary; and FT-DIO also makes weaker assumptions about the operating system's capabilities (TFT requires, e.g., an operating system mechanism to interposition the supervisor between application and operating system, which might not be available on all systems). Moreover, FT-DIO's flexibility and adaptivity features are currently not present in TFT. And similarly to NCAPS, TFT is a warm standby approach as opposed to FT-DIO's hot standby. Therefore, a combination of techniques of TFT and FT-DIO would be very interesting.

Another cluster-based system with similar objectives is Wolfpack, described in more detail in Section 3.3.3. Unlike NCAPS or TFT, Wolfpack uses a cold standby approach and has to restart applications once a failure is detected. FT-DIO uses hot standby, resulting in higher overhead but practically uninterrupted service even in the presence of faults.

FT-DIO also shares some characteristics with the decision module in Simplex [253]. Both observe multiple output data streams and decide which one of them is valid. While Simplex's decision module is specifically geared to facilitate dependable update of running systems by employing analytic redundancy, FT-DIO focuses on improving legacy application dependability by replication. FT-DIO's flexibility with regard to fault model and replication scheme make it possible to implement analytic redundancy as a special case on top of it.

7.2.3 User interface

As a starting point for FT-DIO, consider the typical invocation of a program in a UNIX environment. Program `prog` is started from the command prompt with a number of parameters, and potentially reads input from standard input and writes output to standard output:

```
$ prog arg1 arg2 ...
```

Without any special precautions, this program is susceptible to any number of faults, e.g., crash of the program itself or the machine on which it runs. The solution proposed here is inspired by UNIX's toolbox philosophy: employ small and specialized tools that communicate with each other over well defined and commonly used interfaces. FT-DIO is such a tool; it acts as a wrapper around arbitrary programs, provides fault tolerance to these programs and communicates with them over the standard input/output interface. Fault tolerance is achieved by executing multiple replicas of the original program and coordinating their I/O interactions. FT-DIO is invoked from the command line and takes the original invocation as a parameter:

```
$ FT-DIO prog arg1 arg2 ...
```

A couple of command line switches are used to ensure the proper function of FT-DIO; e.g., the number of replicas can be set or the fault class for the output (or input) data stream can be selected (see Section 7.2.4 for details). With a crash fault assumption for the actual machine, even the machine that serves as the user's terminal is not immune to failure. Since it would constitute an unacceptable single point of failure, FT-DIO has to take precautions against this case: A user can reconnect to an ongoing FT-DIO session, resuming work interrupted by a terminal's crash.

This capability to connect additional terminals introduces the possibility of suspending a running program and resuming work with this program later on another machine.[1] More interestingly, it allows multiple simultaneously active input/output devices for one program. This feature can be used for collaborative work with one program, where an appropriate handler class of FT-DIO is responsible for the input's consistency, or for connecting multiple, possibly faulty data feeds, e.g., sensors, to a program. Therefore, the treatment of a faulty input is necessary as well.

[1] Of course, this is even possible with only one running replica of the program if no fault tolerance is desired.

7.2.4 Fault models and classes

Any real system has two typical sources of failures: the hardware and the software. Hardware failures are unavoidable, but occur rather rarely. Software failures can happen in two essentially different layers: operating system software and application program. For the operating system, one would hope that it, too, fails only very infrequently, but if it does, the consequences are usually just as severe as those of a hardware failure. A pure application-level software failure on the other hand should not compromise the integrity of the operating system. Two levels of fault tolerance can therefore be distinguished: one where the underlying system is assumed to be stable, and only application faults need to be tolerated, and another one where even the basic system environment (hardware and operating system) can fail.

Handling faults on these two levels and preventing them from becoming failures requires two different wrapper architectures: For the application level only the (legacy) application must be endowed with a certain redundancy, and simple, fast, centralized solutions for the wrapper are acceptable. For the system level, even the wrapper must be completely distributed and no single point of failure is acceptable, even though this likely incurs higher overheads.

Within either wrapper architecture, handlers are needed to deal with the actual faults. Faults can be characterized according to the domain in which they appear: the value of data items, their timing, or their order [229]. The fewer assumptions are made about the behavior of the (replicated) application, the weaker the model. Fault models are hence partially ordered according to this weakness relationship. This order is mirrored by a class hierarchy of handler objects, where inheritance corresponds to this weakness relationship. Hence, a derived class has to implement additional functionality to handle the additional possibility of misbehavior allowed by the weaker fault model. In practice, this implementation is done by overwriting some methods in the derived class; details are described in Section 7.2.5.

Inheritance can also be used to tailor the fault handler classes to problem-specific behavior. For example, overwriting a single method suffices to implement voting with limited precision to give some leeway for slight numerical deviations. This handler hierarchy can also be used to implement adaptive fault tolerance: Since all handlers must present a uniform interface to the wrapper, it is possible to switch between different handlers at runtime by exploiting this polymorphism. Implementing adaptivity is then possible by overwriting the copy constructor[2] for a pair of classes. If the handler object decides to up- or downgrade to another fault model, it copies itself to a new object of this particular class and advises the wrapper to use the new handler object henceforth.[3]

The same class hierarchy can be used in both application-level and system-level wrappers. It is usually not necessary to implement different classes for these two cases, but specialized implementations can nevertheless be desirable for efficiency reasons. A more detailed discussion of both fault models and adaptivity can be found in [136].

7.2.5 Implementation issues

Application-level fault tolerance

Implementing the functionality described above is faced with a number of challenges. The input and output data streams of the replicas must be captured and filtered according to the desired fault class; interactive programs should have the illusion of indeed running interactively; and, depending on the fault model, replicas have to be started on remote machines.

For simple, stream-based programs there is a sufficient means to capture the standard I/O streams: fork off a process that executes the command, and redirect the standard input and output of this child process. The

[2] In the manner of speaking of C++, in which FT-DIO is currently implemented.

[3] It might appear necessary to introduce an additional synchronization between handler objects for this handler change. This is not the case. For application-level fault handling, there is only one fault handler object anyway. For the group communication-based implementation of system-level fault tolerance, the Group Communication Layer (GCL) ensures that all (surviving) wrapper processes receive all messages in the same order, including the configuration change messages. Therefore, this synchronization is implicitly performed by the strong order semantics of the underlying GCL.

simplest way to do it is to use two UNIX pipes[4] (one in each direction) between the parent and child processes, created before the fork. The child process, before executing the actual command, duplicates the ends of these pipes to its standard input/output file descriptor. This can be done with the functions `pipe` and `dup2` [278]. Repeating it for a number of replicas represents the simplest case: local replication.

This solution works fine in many cases, however, it is not sufficient for programs that behave differently when run interactively or non-interactively. One example of such a program is `grep`: `grep` processes input directly when run in a shell, but buffers input when run in the background. Therefore, to ensure complete transparency for the user of FT-DIO, it is necessary to run programs as if they were running in a terminal. Pseudo terminals [278] make this possible in UNIX environments, and some software packages offer convenient interfaces. A well known example is Expect [179], which is used in FT-DIO since it is available on many platforms.

Additionally it might be desirable, e.g., for purposes of load balancing or additional fault tolerance, to start replicas, with or without pseudo-terminal wrappers, on remote machines via `rsh`. This gives rise to a second case: distributed replication with standard, unicast communication streams between the replicas and the centralized wrapper.

Programs are hence started locally or remotely, with or without pseudo-terminal wrappers. These wrappers or the programs themselves pass through their input/output to the main wrapper program as sketched in Figure 7.1. This main wrapper communicates with the actual terminal and uses a simple `select` construction to poll from the various file descriptors to accept output from its peer wrappers/applications and the terminal. The wrappers themselves are actually two processes, responsible for either input or output data stream.

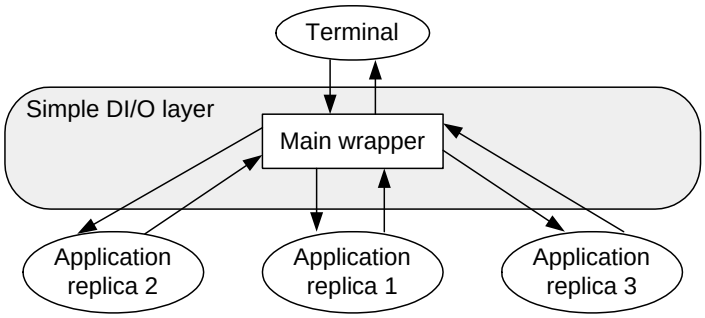

Figure 7.1: Process structure of a simple distributed I/O (without pseudo-terminal functionality). Arrows indicate standard input/output data streams.

While local and distributed, unicast-based replication are simple solutions, they both fall short on one important point: the main wrapper process is still a single point of failure. For truly fault-tolerant distributed I/O, it is necessary to remove this point of failure and to implement its functionality in a distributed manner. The following Section 7.2.5 describes such a wrapper for a system-level fault tolerance.

System-level fault tolerance

For a system-level fault tolerance, no single point of failure is acceptable since any machine might crash. Therefore, it is necessary to replicate the application on remote machines and distribute the functionality of the main wrapper of the application-level solution among all wrappers. This distribution implies a wrapper around every replica, even if no pseudo-terminal functionality is necessary. Figure 7.2 shows the process structure employed by FT-DIO for this distributed replication model.

Implementing the main wrapper's functionality in a distributed fashion requires that every wrapper forwards input and output data to its peer wrappers. This forwarding can be efficiently implemented by a multicast-based group communication protocol. It is important that all wrappers observe all data in the same

[4] Or other, comparable mechanisms of inter-process communication, depending on the underlying operating system.

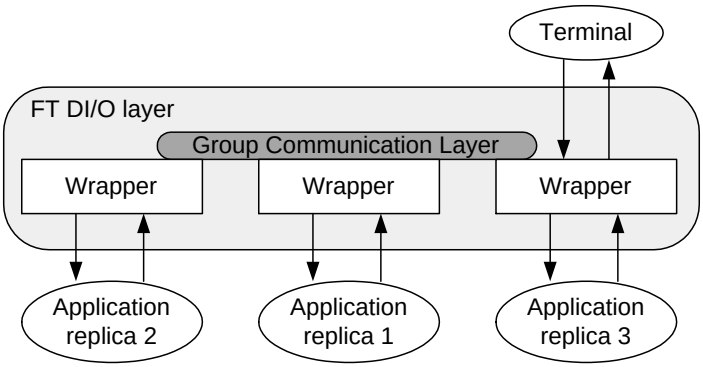

Figure 7.2: Process structure of a fault-tolerant distributed I/O. Arrows indicate standard input/output data streams.

order, otherwise inconsistent decisions could be made by different wrappers. Proper ordering is ensured if the group communication protocol guarantees atomic and total order delivery. Many protocols have been proposed to implement such order semantics; FT-DIO uses an implementation of the Totem protocol [140, 204] as a GCL with the desired properties.

With such an underlaying group communication layer, the wrapper itself becomes rather simple. In each wrapper, there is one handler object for the input and and another one for the output data stream; these are called `InputHandler` and `OutputHandler`, respectively. Two separate objects allow a flexible mixture of fault assumptions for both data flow directions. Figure 7.3 shows the conceptual data flow within a wrapper process.

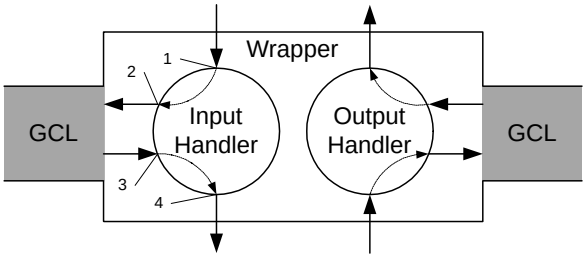

Figure 7.3: Conceptual data flow within an FT-DIO wrapper.

If any input data arrives at a wrapper, the wrapper calls the method `handleData` of the `InputHandler` object (Figure 7.3, Step 1). Typically, the `InputHandler` object processes this input according to its specific policy and eventually hands the data on to the GCL by means of a call to the method `sendData` (Step 2 in Figure 7.3). Later, this data as well as input from other wrappers will arrive at the `InputHandler` via the GCL (Step 3 in Figure 7.3) as an argument of method `receiveData`. Again, the `InputHandler` decides what to do with the data, e.g., to pass it on to the controlled application process by calling `writeData` (Step 4 in Figure 7.3). The same procedure applies, mutatis mutandis, to the `OutputHandler` object.

The wrapper must never block during a communication call, e.g., when trying to read from or write data to a controlled program that is not responding. To ensure such non-blocking behavior, the wrapper uses an extended `select` construction (see Figure 7.4). The wrapper has to check for data being available from the GCL, a potentially present terminal, and the controlled process. It also has to check for the possibility to write data to the terminal or the controlled application process, if the input or output handler want to send data. For this reason, the handler objects must not write data directly to the corresponding sockets, but call `writeData`. Additionally, a listen socket has to be supervised to enable new terminal connections. This ability to attach and detach terminals to/from the wrappers also allows moving the terminal window of an

application to another machine, without the application even being aware of it.

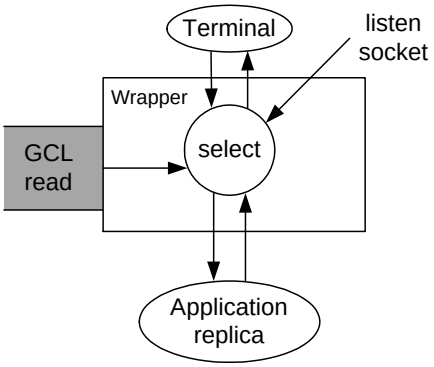

Figure 7.4: `select` loop of an FT-DIO wrapper process.

The class hierarchy

The hierarchy of fault models from Section 7.2.4 is mirrored by FT-DIO's fault class hierarchy. At the bottom of this class hierarchy is an abstract class `FTDIO_BaseIOHandler`. This class provides a common interface description as well as some basic methods—e.g., methods to determine whether it is safe to shut down a wrapper because the controlled application has terminated or if there is still unprocessed data. Another such method is `ConfigChange` that handles configuration change messages of the GCL. This method is also used for adaptive fault tolerance as described in Section 7.2.4. These methods are internal to FT-DIO and a developer usually does not need to concern himself with these functions.

Derived from the base class is `FTDIO_None`, which implements only trivial, non-fault-tolerant functionality. Fault tolerance can be added by further subclassing from this class and overwriting the virtual methods `handleData` and `receiveData`. Additionally, `sendData` and `writeData` could be modified, but this is rather atypical. Usually, `handleData` just calls `sendData`, and the actual fault-tolerance algorithm is implemented in `receiveData`. E.g., `FTDIO_Crash` counts the amount of data sent from each replica and picks the fastest sender, and `FTDIO_Comp` stores data from all replicas and invokes a voting function to decide which is a correct value (tolerating computational faults). To support varying voting schemes, `FTDIO_Comp` provides a virtual `vote` method and data buffers that are automatically (de-) allocated in case of configuration changes. The method `vote` can be overwritten to implement early decision voting, resulting in `FTDIO_Crash_Comp` (tolerating both crash and computational faults), and a supplementary method `vote_item` is available to implement voting on different data types. Similarly, handlers can be implemented to take care of output as produced, e.g., by multi-version programmed software than can be different, but nonetheless correct. More generally, handlers can be easily fitted as close to an application as desired, cleanly separating actual application semantics and corresponding fault-tolerance mechanisms.

If objects from this class hierarchy are used in a wrapper context from Section 7.2.5, the wrapper just shortens Step 2 from Figure 7.3 to Step 3. However, a more efficient class specifically geared toward application level fault tolerance could directly call Step 4.

7.2.6 Some experiments

Some initial measurements for a prototype implementation of FT-DIO are provided in this section. To assess FT-DIO's overhead, the simple UNIX command `cat` was used to copy one file to another. `cat` as such is not a typical candidate for replication, but since its own overhead is very small, it allows to isolate the overhead introduced by the FT-DIO layer. Additionally, since `cat` is highly I/O-bound it represents the worst case for a system like FT-DIO. The environment consisted of three AMD K6 based PCs running at 300 MHz under Linux with kernel version 2.0.36, connected by a switched 100 MBit/s Ethernet.

All three implementation variants (local replication, distributed replication with unicast messages, and distributed replication with multicast-based group communication) were used to implement three different levels of fault tolerance: no fault tolerance at all (one replica) to serve as a base case (see Figure 7.5), tolerating crash faults with two replicas (see Figure 7.6), and tolerating computational faults via voting with three replicas (see Figure 7.7). The handlers were optimized to fit with either application-level or system-level fault tolerance (without pseudo-terminals). Files of 128 KBytes, 512 KBytes, and 1024 KBytes were copied to/from the local hard disk of one machine, and all reported numbers are averaged over 20 runs.

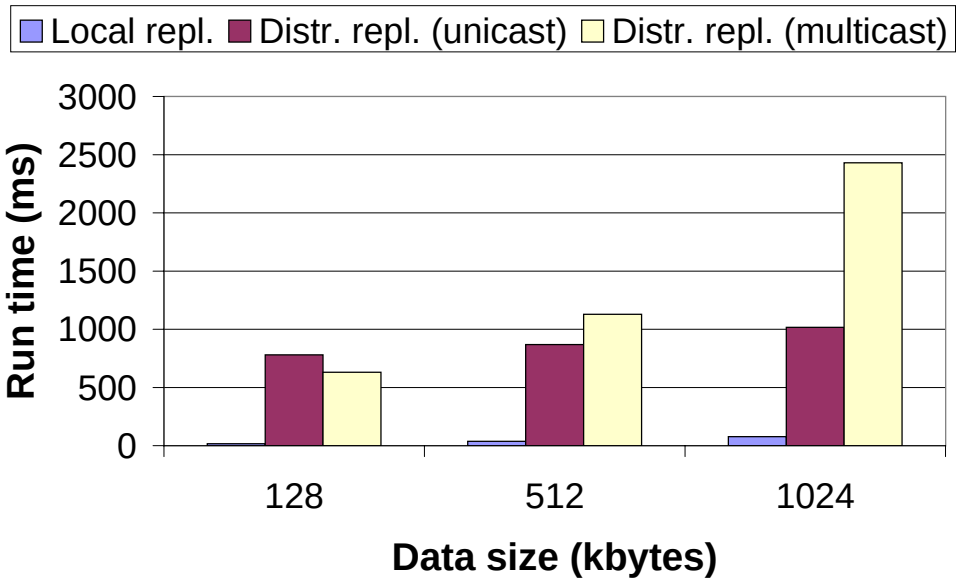

Figure 7.5: Average runtime of `cat` under FT-DIO control, shown for different data sizes and replication schemes, one replica.

Two main conclusions can be drawn from these results. First, if only software fault tolerance is desired for simple fault models like crash fault, local replication is vastly superior and the overhead of replication to remote machines is unaffordable (but note that the test program was I/O-bound and not CPU-bound). This changes, however, if more sophisticated fault models like computational fault are to be considered. Under such assumptions, the additional overhead of distributed replication is small compared to the overhead introduced by voting (the numbers for local replication and unicast-based distributed replication in Figure 7.7 differ only slightly). Second, completely removing a single point of failure incurs a considerable overhead. But if machine crashes and sophisticated fault models are to be considered, this price might become affordable. Obviously, it depends on the application scenario whether or not a single point of failure in a small, controllable piece of software is acceptable—in particular since hardware faults are relatively less frequent than software faults [95].

7.2.7 Conclusions

The FT-DIO system has been proposed as a solution for using replication to improve the dependability of legacy applications. Motivated by the fact that existing applications cannot be modified at acceptable cost, a wrapper approach based on the interface-observable behavior of an application has been chosen; the proposed FT-DIO system implements this approach.

For both application-level and system-level fault-tolerance requirements, appropriate wrappers have been designed. Embedded in these wrappers are objects that implement the actual fault tolerance, based on the intercepted data streams of the application. These objects are instances of a hierarchy of fault classes that closely mirrors the hierarchy of fault models. By subclassing from a given fault handler class, handlers for new fault models can easily be implemented. This hierarchy allows easy customization at start time, as well

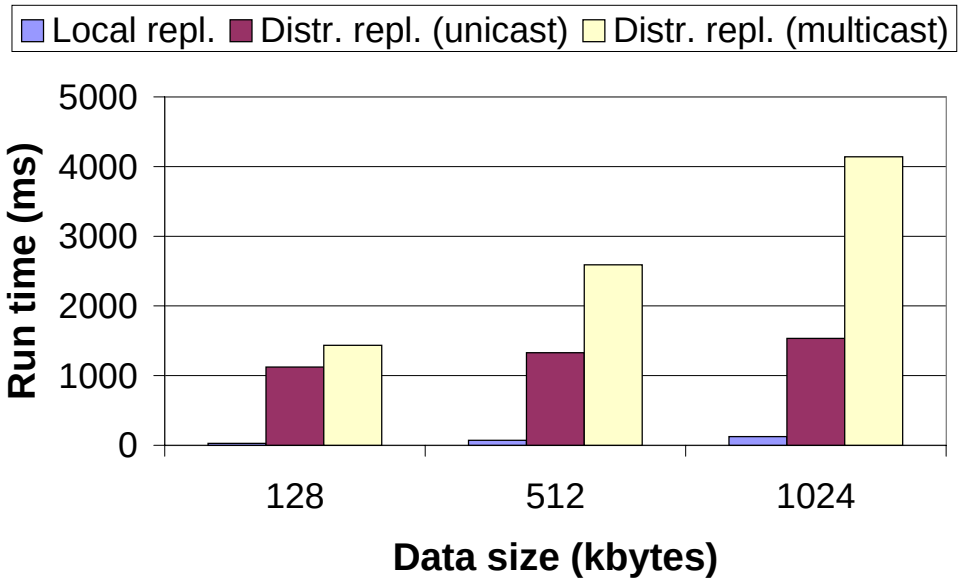

Figure 7.6: Average runtime of `cat` under FT-DIO control, shown for different data sizes and replication schemes, two replicas, tolerating crash faults.

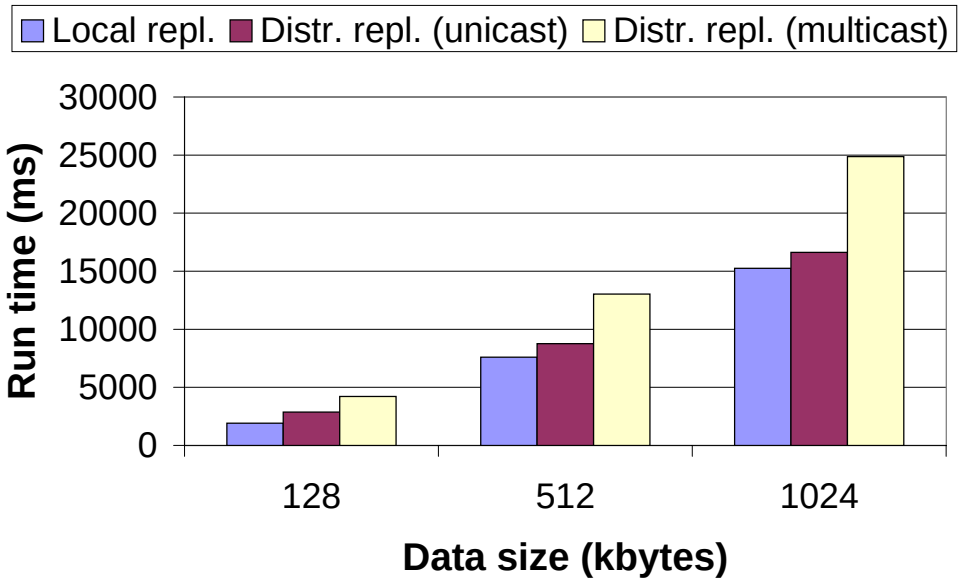

Figure 7.7: Average runtime of `cat` under FT-DIO control, shown for different data sizes and replication schemes, three replicas, tolerating computational faults.

as adapting the fault-tolerance level at runtime. Moreover, FT-DIO can avoid any single point of failure, including a user's machine, by allowing a user interface to reconnect to an ongoing application. FT-DIO can also be used for new applications by separating fault-tolerance functionality from the actual program semantics.

FT-DIO provides replication based on the standard input/output interface, resulting in flexibility tradeoffs different from those in systems like TFT, NCAPS, or Wolfpack: FT-DIO makes more stringent assumptions about the interfaces used by the application, but allows a large flexibility with regard to fault models, replica behavior, cold, warm, or hot standby, and some degree of nondeterminism as potentially introduced by, e.g., analytic redundancy.

Measurements show that for local replication, the overhead is small for simple fault models like crash fault. For advanced fault models like computational faults, which requires voting, even the distributed implementations that have no single point of failure become competitive.

7.2.8 Possible extensions

Completing the fault handler class hierarchy to include other fault models than the ones described above is mostly interesting for complicated models like Byzantine faults. Such a comprehensive class hierarchy would allow a comparison of the overheads necessary to deal with various fault models in a uniform, yet practical environment.

The approach taken here is that of using software as building blocks with standard interfaces. Plugging such blocks together results in new blocks with enhanced properties. It is an interesting and practically relevant, yet challenging research question what other properties beyond fault tolerance can be achieved by composing such software blocks. Examples for such properties include security and certainly responsiveness.

On a more practical side, interfaces other than standard input/output streams could be considered, e.g., network connections. This is related to research projects that target a virtualized operating system where the location transparency of applications and user interfaces is a main objective. An example for such a project is the Computing Communities project at New York University. The methods proposed here allow such systems not only to achieve location transparency, but also to provide fault tolerance.

Such an extension would also address the limitations of the approach used here. Any application that modifies some permanent state by means other than the standard I/O channels would not behave consistently under FT-DIO. FT-DIO extended in this sense would then likely converge with systems like TFT.

7.3 An experimental investigation of group communication

7.3.1 Introduction

In the previous Section 7.2, FT-DIO, a wrapper-based solution for handling the input/output of replicated application processes, has been introduced. In its most powerful form, FT-DIO uses the Totem group communication protocol [5, 204] to implement multicast communication between the replicas in a consistent and efficient manner. Group communication protocols in general are a popular tool for building distributed systems (as has been discussed in Section 3.3.2).

For the use of FT-DIO as a responsive system, not only the logical consistency of such a group communication protocol is relevant, but also its behavior with regard to faults and real time. One reason to choose Totem as message layer for FT-DIO is that Totem supposedly has a low, predictable message latency [204]. In particular, MOSER and MELLIAR-SMITH [203] give an analysis of Totem's message latencies for a very specific fault model.

Traditionally, distributed real-time applications are often built using preplanned schedules for message transmission—the Time-Triggered Protocol (TTP) [151] is a typical example (see Section 3.5.3). Event-triggered systems like Totem are supposed to work better in complex environments (such as represented by FT-DIO) where a preplanning of all activities is not possible while nevertheless providing message latencies

that are below a given bound with very high probability. Therefore, for an investigation of such a protocol with regard to responsiveness, the distribution of end-to-end message latencies is the metric of choice.

In this section, the results of MOSER and MELLIAR-SMITH [203] are compared to actual measurements of the timing behavior of Totem under an extended fault model. Some additional related work is briefly mentioned Section 7.3.2, the Totem protocol in particular is described in some more detail in Section 7.3.3. The models used for the experiments are introduced in Section 7.3.4 and the experimental results are presented in Section 7.3.5. In Section 7.3.6, the experiments and their relation to analytical results and Totem's suitability for responsive systems are discussed. Finally, the discussion of Totem is summarized in Section 7.3.7 and possibilities for future work are considered in Section 7.3.8. The investigations and conclusions presented here are the result of joint work with M. Werner and L. Küttner; additional details can be found in [137, 140, 141, 156].

7.3.2 Related work

The basic ideas of group communication have already been described in Section 3.3.2. In the context of this section, group communication protocols with analytical or experimental evaluations of timing properties are of particular interest.

Mainly motivated by the need for a simple programming model for distributed real-time systems, RAJKUMAR et al. [236] describe the real-time publisher/subscriber interprocess communication model. Processes can publish messages, without being aware of the number of receivers, and receivers can subscribe to any number of messages. The implementation of this model is portable and analyzable and has been extended in [235] to tolerate processor failures as well.

The RTCast system [1] is designed to provide delivery of both periodic and aperiodic group messages in bounded time and attempts to combine the flexibility of event-triggered approaches with the low delivery times of time-triggered systems. To be able to achieve this combination, a private Ethernet network is assumed (making media access deterministic since RTCast itself is token-based); processor failures are considered, but upon message transmission errors the receiving processor shuts itself down (an extension with a bounded number of retransmissions is alluded to). This treatment of lost messages limits the practical usefulness of RTCast.

The latency of Totem messages has been analyzed for a very specific fault model [203]. An experimental study of Totem [54] is only concerned with performance but not with fault injection. A comparative experimental study [249] of Totem and two similar group communication systems has shown that Totem is particularly suitable for environments with many simultaneously active senders, while it is outperformed by simpler protocols if only relatively few nodes want to send messages concurrently. Since FT-DIO is such a multi-sender scenario, these results corroborate the choice of Totem for FT-DIO.

7.3.3 The Totem protocol

The Totem protocol is structured in several subprotocols. On top of a best-effort multicast communication service like User Datagram Protocol (UDP), the single-ring protocol provides reliable and totally ordered broadcast services for a single LAN. This layer also takes care of configuration changes. The multiple-ring protocol is used to interconnect several LANs, offers a globally ordered multicast and monitors the topology of this network. On top of the multiple-ring protocol, the process group interface provides the abstraction of several independent process groups running over this group communication system. Only the single-ring protocol's behavior is considered here.

Totem's single-ring protocol realizes group communication by using a token-based multicast. A process is only allowed to transmit a message if it owns a rotating token. This token must be passed on to another process after a certain time; during this time the owner may multicast a predetermined number of messages. Once a message is received by a process, it is only delivered to the application if the order constraints are fulfilled: all prior messages have been delivered, and all other processes have also received the message.

In case a node has not received a message (this can be decided using information in the rotating token), a retransmission request for this message is stored in the token. This request is handled by any node that has already received this message. To protect against token loss, the token is resent after every Token Retransmission Timeout (TRT) until a new message or token is received or until the Token Loss Timeout (TLT) occurs. A lost token is considered as an indication of a configuration change and the membership protocol is initiated.

The membership protocol processes configuration changes (nodes joining or leaving a group) and has two important objectives: all members of a new configuration must reach consensus on which nodes are group members and the protocol must terminate in bounded time. In particular, the Consensus Timeout (CT) gives a maximum time for processors to reach this consensus. If this timeout has occured, those processors that did not reach consensus are marked as faulty and ignored for the remainder of this round. In an extreme case, exclusion of processors continues until single processor configurations are reached. Obviously, the values of TRT, TLT and CT are critical for Totem's performance. A more detailed description of the Totem protocol can be found in [5, 204].

7.3.4 Models for experiments

For the following experiments, the system is assumed to consist of a number of workstations that are connected by a standard Ethernet LAN. The workstations are dedicated to the application, the network can have additional traffic. Owing to the nature of an Ethernet network, messages can be lost or delayed; the network is capable of unreliable multicast within a single LAN.

The Totem layer is used by only one distributed application with one process per machine. These processes generate messages according to a Poisson arrival process. Faults are injected in the communication sublayer and in the processes themselves (since there is one process per processor, they can be identified in the present context). Individual messages are randomly lost following a Bernoulli distribution, and burst errors in the communication layer are simulated with Poisson-distributed arrival times and log-normal length distribution; messages are always delivered to the originating process.[5] Processor faults are modeled with intervals in which no messages are processed (similar to a burst error, but with different parameter settings). Additionally, external load on the machines is considered. A more detailed discussion of these model assumptions can be found in [140, 156].

7.3.5 Some experiments

To investigate the suitability of Totem for responsive systems, the message latency is used as a proper metric. This latency is measured from a start of sending a message to a finish of receiving the same message. Since Totem's safe delivery mode is used, receiving a message in the sending process implies that all other processes also have received this message. The measurements were carried out on eight workstations,[6] connected by a 10 Mbps Ethernet. Messages of 1024 bytes were generated on each host by a Poisson process with arrival rate 62.5 messages per second, resulting in a total of 500 messages per second on average. One experiment consisted of transmitting 5000 messages per process.[7] Totem's timeout values were set to TLT=400 ms, TRT=40 ms, CT=300 ms, and JT=40 ms.

Based on these parameters, the message latency distribution without fault injection, with single message faults for receiving messages, and with external load (a compiler process on one machine) is shown in Figures 7.8 to 7.12. Additional information about the experiments can be found in [140], where also the impact of bursty message loss, message loss during sending messages, and simulated processor crashes is discussed in more detail.

[5] Poisson processes are commonly found when modeling the random arrival of tasks, clients, etc. [283]; the Bernoulli distribution corresponds to a random, independent decision of success for every individual message; and the log-normal distribution is commonly used when events (here the length of the downtime) with strictly positive parameters are to be modeled.

[6] Four SGI IP22 running IRIX 5.3, two Sun SPARC20 and two Sun Ultra 1 running Solaris 2.5.

[7] The large number n of samples x_i allows a fairly good approximation of the (unknown) true distribution by the empirical distribution $S_n(x)$ according to a Kolmogorov-Smirnov fitness test [103, p. 240] as discussed in more detail in Section 6.6.2. For $n = 5000$, the true distribution differs less than 2% from the empirical distribution at a confidence level of 95%.

Figure 7.8 shows the latencies for 5000 messages without fault injection. The vast majority of latencies is below 60 ms, but some outliers are obvious. This is conjectured to be due to scheduling delays caused by system processes and will be discussed further.

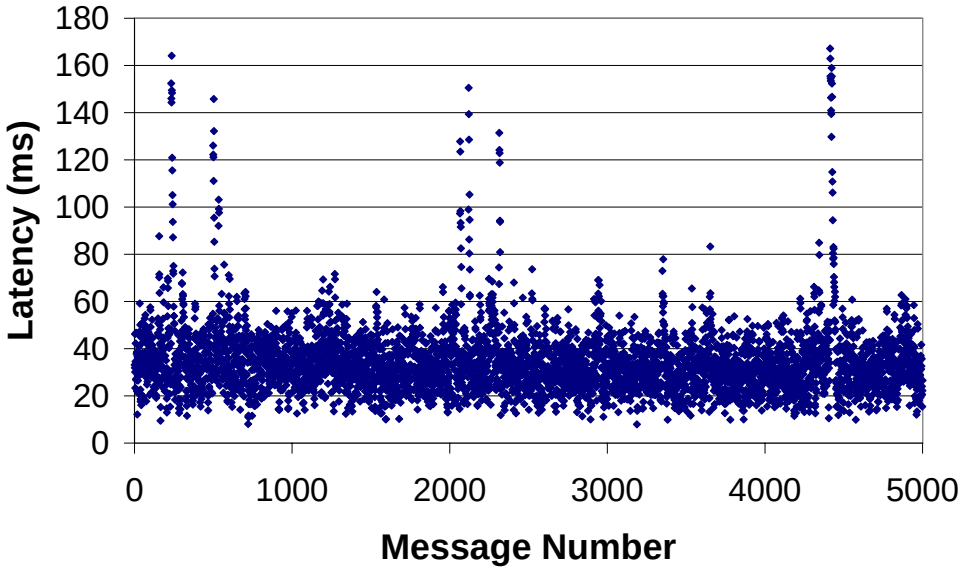

Figure 7.8: Totem message latencies without fault injection or additional load.

Message reception loss (as analyzed by [203]) is modeled by a Bernoulli process with varying reception loss probabilities p_{nr} (nr for "not received"), Figure 7.9 shows the results for $p_{\mathrm{nr}} = 0.01$—note the higher maximum latencies and the smooth spreading of latencies. Figure 7.10 shows the distribution for varying p_{nr}; mean latencies are shown in Table 7.1.

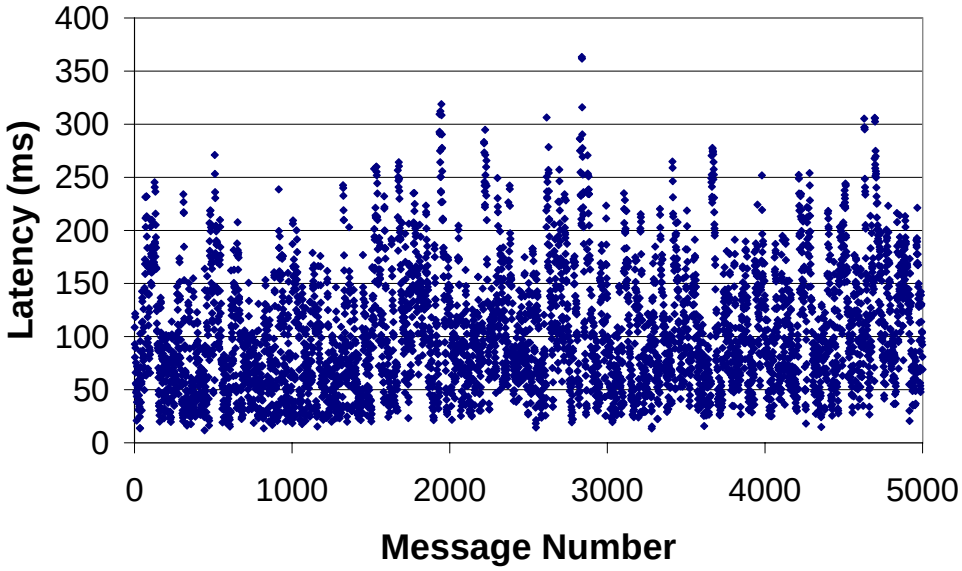

Figure 7.9: Totem message latencies with $p_{\mathrm{nr}} = 0.01$.

How does this compare to a real, large scale source of scheduling delays? To address this, a compiler is run as a background load on one of the nodes participating in the group communication; the results are shown in Figure 7.11. Notable in Figure 7.11 are the sudden large differences in the latency—particularly if the group communication's membership protocol is started due to a suspected configuration change.

Combining all possible faults scenarios (message loss both single and in bursts, simulated processor crashes, and background load) is a major challenge for any communication layer. Figure 7.12 shows the

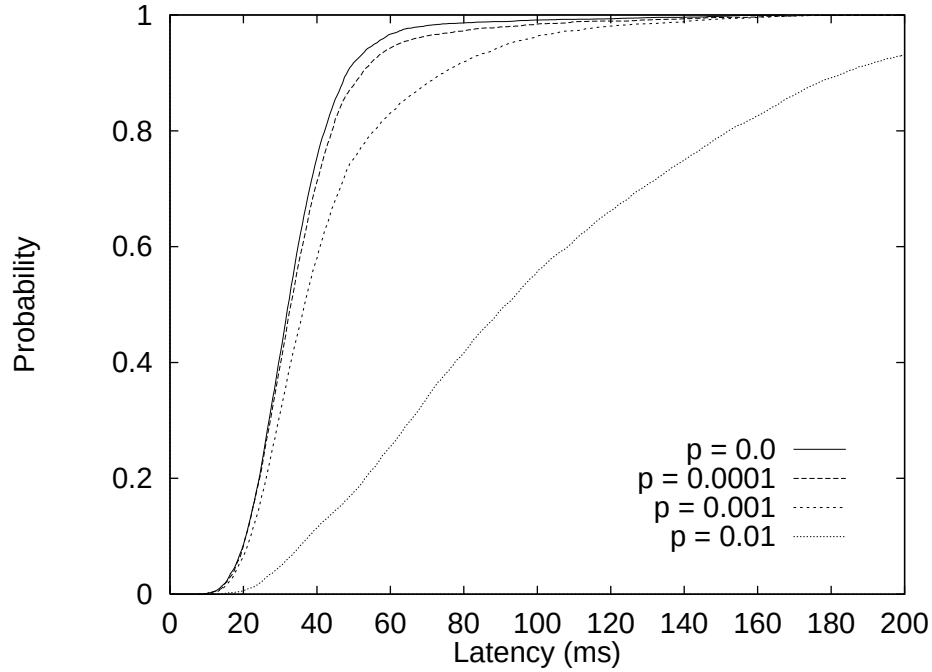

Figure 7.10: Probability distribution of Totem message latency for varying p_{nr}.

p_{nr}	mean latency
0.0	34.38
0.0001	36.30
0.001	43.38
0.01	103.50

Table 7.1: Mean latencies of Totem messages for varying p_{nr}.

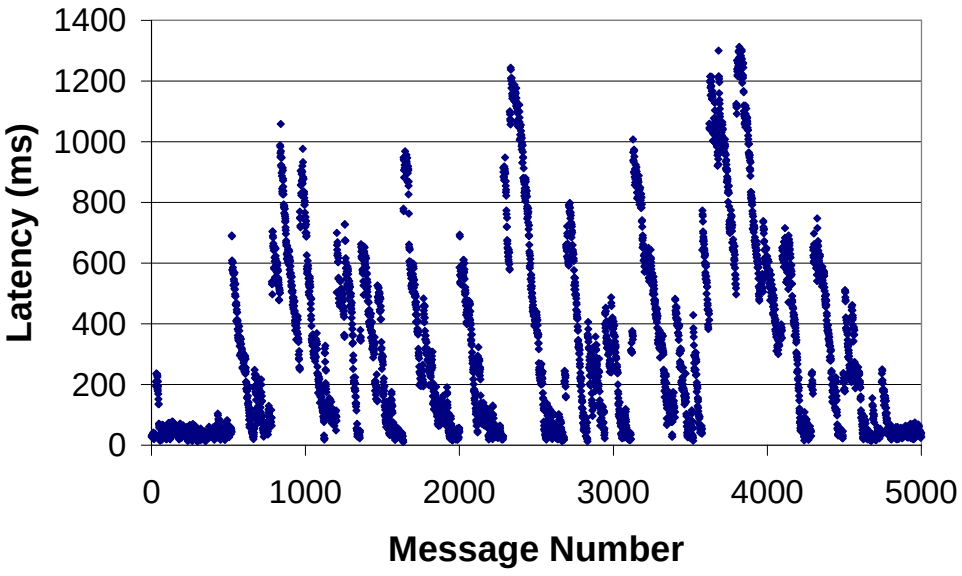

Figure 7.11: Totem message latencies with a compiler on one machine.

latency distribution for the combination of all these factors and also gives an overview of the latency distribution for some of the individual experiments mentioned above.

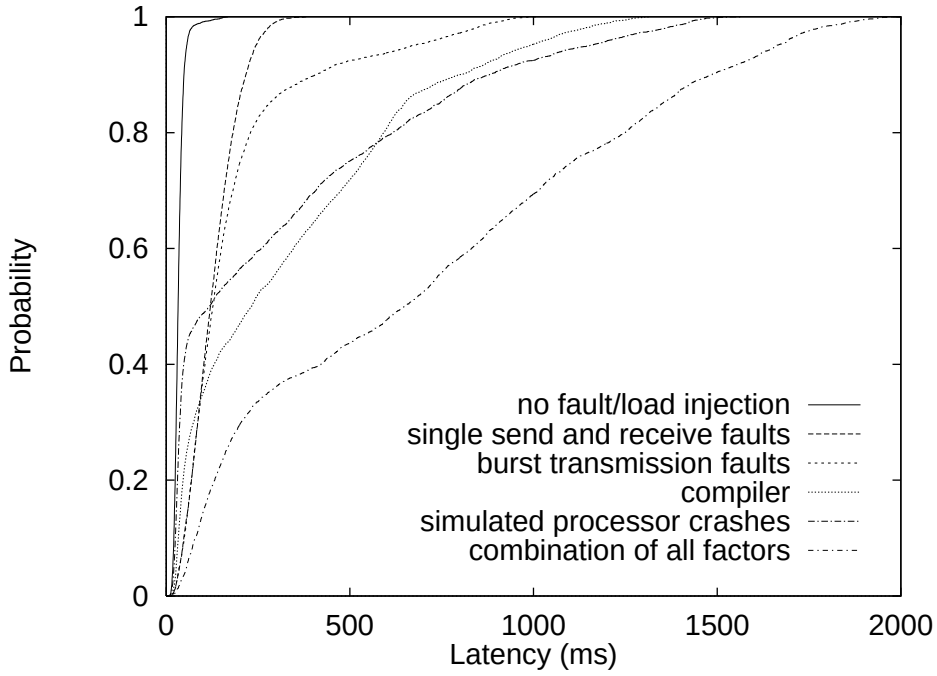

Figure 7.12: Overview of several Totem experiments—probability distribution of Totem message latency.

7.3.6 Theory and practice in Totem

How do analytical and experimental results for message latencies in Totem relate to each other? The experimental results are naturally not as general as an analysis. On the other hand, they are not burdened by the need to simplify the actual system to make it tractable, but rather reflect reality with all hidden complexities. The simplifying assumptions in the analysis of Totem do not allow to reflect effects that are relevant in practice. Some of these simplifications are: the fault model only considers message loss during reception (an UDP protocol stack is always allowed to discard messages, even during sending), the token itself is never lost, and the flow control mechanism does not interfere with sending of messages (making it impossible to use the entire available bandwidth). Based on such assumptions, MOSER and MELLIAR-SMITH [203] claim a predictable behavior of Totem.

Comparing this analytical result with the experiments that deal with message loss basically confirms this claim. As long as no other faults or background load are present, Totem indeed shows a reasonably predictable behavior. However, some of the quantitative aspects do not match too well between analysis and experiment. For example, the analysis predicts a sharp increase of message latencies if a certain threshold fault rate for message reception is exceed. This increase does indeed happen (see Table 7.1), but the threshold is smaller than what the analysis would let one expect. This should—and can—be accounted for with careful calibration of Totem's critical parameters TRT, TLT, and CT.

The experiments go beyond the assumptions of the analysis. In particular, scheduling delays were found to be of crucial importance for the message latencies. This fact is important since in a COTS environment, no control over the system load or scheduling behavior can be assumed a priori. Such scheduling delays can even result in initiating the membership subprotocol—even though no process has actually failed or left the group—and in cascading of message delays (e.g., between messages 2400 and 3000 in Figure 7.11). This cascading also implies that message latencies are difficult to predict in the presence of a processor crash. These problems reiterate the need to carefully choose Totem's timeout parameters, depending on factors like

system load.

Extending Totem's analysis to take into account such effects as scheduling delays and (even cascading) membership protocol invocations would be necessary to truly judge Totem's theoretical response time distribution. The experiments indicate a need to carefully control the execution environment of a Totem-based application if latency guarantees are needed.

Totem does an admirable job to survive even concentrated fault injection. But the mean latency increases from 35 ms in the simplest case to about 695 ms in the most complex scenario; worst-case latencies can reach up to 2 s. To use Totem in a responsive system, conservative assumptions about both the fault model and the message latencies are necessary, along with careful adjustment of Totem's parameter to the actual system and application.

7.3.7 Conclusions of Totem experiments

Experiments for the message latencies in the Totem group communication protocol under different load and fault scenarios have been presented in this section. For simple fault and load scenarios, the predictions of Totem's theoretical analysis are essentially confirmed: Totem does behave predictably.

Under more complicated fault scenarios, or even under background load on some of the nodes, this is no longer the case. Totem suspects configuration changes and initiates the membership protocol, resulting in longer and much more difficult to predict latencies, even if the configuration has not actually changed. To account for these unpredictabilities, an analysis including Totem's membership subprotocol is needed, as also suggested by MOSER et al. [204].

The results of this section show the need for a careful investigation of an application's actual communication requirements. They also question the usefulness of event-triggered protocols like Totem for real-time applications—compared with time-triggered protocols like TTP—since the probability of message delivery before a deadline can be substantially reduced due to faults in an unpredictable way. On the other hand, Totem is certainly more flexible than TTP as the latter assumes all failure modes and load profiles to be known a priori.

Since there is a lack of analytical results for the general case, the question of Totem's predictability in faulty environments remains open. However, in environments were guarantees on the execution can be given and the message fault rate is low, Totem is a good example for a responsive group communication system.

7.3.8 Possible extensions

The need for an extended analysis of Totem has already been pointed out. Additional experiments with deterministic message generation or with a feedback mechanism from the group communication layer to the application (allowing the application to adapt to the current latencies) should provide additional insight into the behavior and usefulness of the Totem protocol. Also, the integration of controlled user-level scheduling (following a concept as proposed in Chapter 8) with the Totem protocol should allow vastly improved predictability in the presence of background load. Such a controlled scheduling would also allow a better adaptation of timeout values in the Totem protocol to the actual environment.

7.4 Replicating the Calypso master

After the problem of input/output has been addressed in Section 7.2, it is possible to abstract away from it and concentrate on replicating the Calypso master itself, focusing on the parallel steps. Ideally, a replicated design should consider both the improvements in fault tolerance and the opportunities for increased performance. Design choices are discussed in Section 7.4.1, some implementation issues are presented in Section 7.4.2, and some experimental results are shown in Section 7.4.3.

7.4.1 Design options

With multiple master processes, the problem of keeping them consistent arises again. As has been the case in FT-DIO, multicasting messages to all masters is an appropriate solution, since it also allows a rapid dissemination of memory updates generated by worker processes (especially if the communication network supports multicast). The question here is how to arrange the master and worker processes in multicast groups.

One possible solution is to include all masters and workers in a single group. While this would allow workers to "sniff" the results of memory requests posted by other workers and hence to reduce page faults, the overhead imposed by unrelated messages can be considerable. Another solution is to use one group per worker, including this worker and all masters. This seems a more viable approach: masters could react on a per-message basis to requests by a worker and dynamically share the load of handling these requests. However, this load sharing either requires synchronization among the masters (to decide which master answers a particular call) or increases network load considerably if brute force methods (all masters answer all requests) are used.

Alternatively, all masters can be placed in a single group, and workers are not a member of any group at all (somewhat similar to the "open group" of [129]). Such a structure implies that messages to a worker process cannot be sent via the multicast sublayer, but must use unicast connections. To establish this connection a worker contacts a master that will take care of this worker's requests. But memory updates generated by a worker can be sent via the multicast layer to all masters without any need for synchronization among each other. Therefore, results computed by workers are available to all masters.

If load sharing among the masters becomes necessary, they can instruct workers to connect to other masters. This can be based on local decisions (a master detects that it is overloaded) or by additional consensus among the masters. The advantage of this design is that the results of this consensus have no impact on the execution's correctness. The mechanism for redirecting workers to other masters can be identical to that used for protection against master crashes: workers detect closed communication channels and autonomously connect to other masters. Masters can remain completely oblivious of each other but can also decide to exchange status information at their discretion. There is no need for synchronization among the masters, not even at the end of a parallel step: A master will only complete a parallel step if completion messages and memory updates have been received for all routines. But since these messages arrive at all masters (if a reliable multicast is assumed), all masters are assured to receive results of all routines in a parallel step.[8]

The assumption of reliable message delivery is indeed the only necessary assumption about the message layer. There is no need for even FIFO semantics in the communication layer, since Calypso's programming model explicitly does not predicate any order of the execution of parallel routines.

This single group design is especially suited to large configurations with many workers. For the experiments presented below, only four machines were available. It can therefore be expected that with regard to performance, replication is inferior to both plain Calypso and even Calypso with checkpointing. Also, the read/write ratio of a program has some influence on the performance of the program with replicated masters, since reads are shared among masters but writes have to be distributed to all of them. Hence the larger the read/write ratio, the better Calypso with replicated masters should perform.

7.4.2 Implementation issues

An advantage of the design described last in Section 7.4.1 is that it fits very well with the Calypso system in general; especially the absence of any explicit communication between the masters. Masters handle their assigned workers identically to the non-replicated case. If routine completion messages from other workers arrive via the group communication layer, the master only has to check if they belong to the current parallel step and can then include these messages into the result list for this parallel step. From the master's perspective, such a result appeared out of nowhere.

[8] A slow master can store completion messages for parallel steps that it has not yet reached and can later use these results immediately.

It proved to be slightly problematic to find a group communication system that implements a very simple messaging semantics, namely reliable delivery. Many of the already described group communication protocols go to great lengths to efficiently implement complicated semantics, but only a few systems provide flexibility in choosing the desired semantics or offer simple semantics at all.

One such system with flexible semantics is Horus [289], but it is not available for the Linux operating system that was used for the Calypso experiments. AMIR and STANTON [6] describe the Spread system. Spread provides all necessary semantics and is also compatible with a Linux environment. Since Spread targets both local and wide area environments, it uses a daemon-based implementation that does not promise particularly good performance.[9] Spread was therefore used as a practical compromise. However, it is only an issue of coding effort to use other, potentially more efficient group communication implementations.

7.4.3 Some experiments

In assessing the performance of replicated Calypso, the same program was used as for the checkpointing experiments presented in Section 6.6.2: twenty consecutive parallel steps of 1 s runtime each, implemented with varying granularities. Four worker processes were used, along with one, two, or four replicas of the master process (even a single master uses the group communication to account for all overheads and to better compare it with the numbers for plain Calypso from Section 6.6.2).

Here the same statistical problem appears as in the checkpointing experiments: To give statistically relevant numbers for many different parameter settings would require experiments running over many months. Therefore, some preliminary experiments were performed with 100 repetitions each to identify parameter settings that appear reasonable. For these settings, the experiments were repeated with a larger number of runs to produce confidence bands around the empirical distributions of maximum width 5% at a confidence level of 95% (as described in Section 6.6.2).

In preliminary experiments, granularities smaller than 10 ms proved to be too small to produce any satisfying performance, owing to the much larger communication overhead. Among the more closely inspected granularities of 10 ms, 50 ms, and 100 ms, 50 ms turned out to produce the best performance—Table 7.2 shows the average runtimes for this experiment, averaged over 100 repetitions (the last line shows the corresponding times for the plain Calypso version without either checkpointing or replication). These results indicate that 50 ms is a good granularity choice for this program in this particular environment.

	Granularity (ms)		
Number of replicas	10	50	100
1	10.33	9.87	11.76
2	12.19	10.46	13.12
4	17.29	13.48	15.38
(plain)	9.37	9.09	9.06

Table 7.2: Average runtime of Calypso program with varying granularity and number of replicated masters, no fault injection. Last line shows times for plain Calypso without replication support.

Table 7.2 also indicates that for this experiment, load balancing among the masters does not compensate for the additional overhead imposed by the group communication layer: the single master is always faster than replicated masters, and the plain Calypso implementation is the fastest version. As has been discussed in the design description, multiple masters can only be expected to improve sheer performance for large numbers of

[9] Additionally, late during the implementation of the replicated Calypso master, some bugs in Spread's scatter/gather communication interface became apparent. While it was simple to circumvent them by copying data into separate buffers and using the normal communication interface, this incurs additional runtime costs, further hampering the performance.

workers. As illustration of the program's behavior, Figure 7.13 shows the runtime distribution for granularity 50 ms and 1, 2, or 4 master replicas.

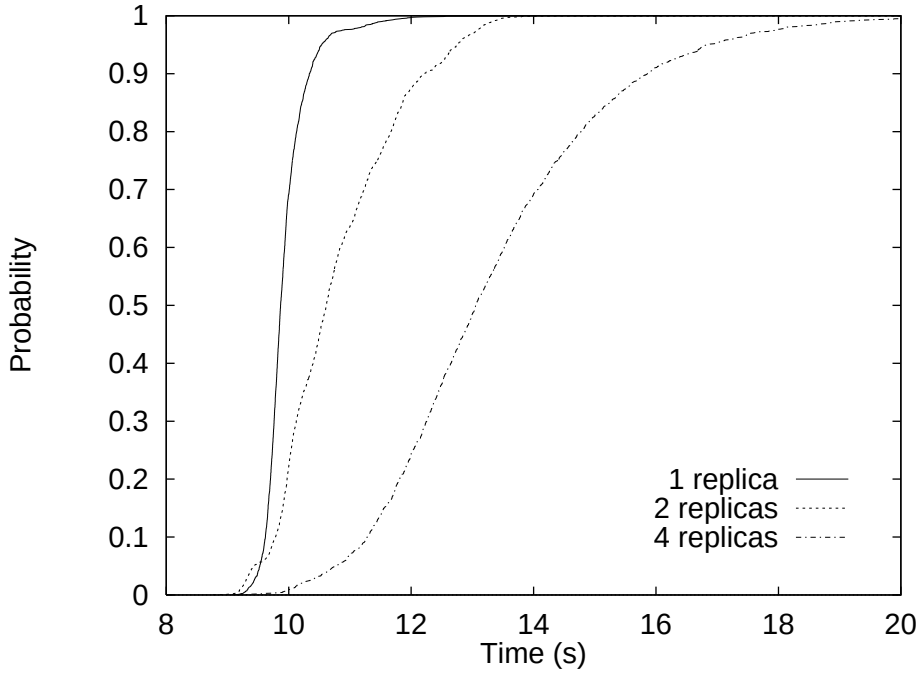

Figure 7.13: Runtime distribution of Calypso test program with varying number of masters, 50 ms granularity, no fault injection, confidence bands narrower than 5%.

The larger number of masters, however, improves the probability of completing the program if faults are injected in the masters.[10] Results are shown here for the same fault rates that were used in the checkpointing experiments: Figure 7.14 with mean master lifetime 20 s, Figure 7.15 with 50 s, and additionally mean master lifetime 200 s in Figure 7.16; the granularity was 50 ms in all experiments reported here.

Considering the very high fault rates, it is not surprising that replication alone is not sufficient to guarantee deadlines, let alone to ensure that the program will eventually finish: For four master replicas and 20 s mean lifetime, the probability of never completing the program is about 3.8% (Figure 7.14), for 50 s mean lifetime, it is 0.7% (Figure 7.15); both numbers are based on the value estimate of the success probability. When using a mean lifetime of 200 s and four replicas, all 1700 experiments actually completed successfully. Additionally, the responsiveness (at a deadline of 16 s) is in both cases (20 s and 50 s) lower than with checkpointing—Table 7.3 shows value estimates of the responsiveness as well as lower and upper limits of the Pearson-Clopper confidence interval (at a confidence level of 95%) for a mean master lifetime of 20 s, Table 7.4 for 50 s (compare these two tables with Table 6.3 and Table 6.4 on page 93, respectively, to see that replication is inferior to checkpointing with a proper choice of the checkpointing interval) and Table 7.5 for 200 s. For four replicas, a mean lifetime of 20 s appears to have higher responsiveness than the one for a mean lifetime of 50 s, but this higher value is not statistically relevant at a 95% confidence level (as tested with a χ^2 test with

[10] An analytic treatment of this problem would start out by generalizing Equation (5.3) on page 69 to include a replicated master. For k replicas, this would result in

$$\Pr(\bar{Z} \leq t) = \int_{\tau=0}^{t} (1 - (1 - \Pr(S_{\text{master}} > \tau))^k) f_Z(\tau) \mathrm{d}\tau \tag{7.1}$$

if master processes have independent, identically distributed lifetimes. However, this does not take into account the overhead caused by the replication (e.g., administrative overhead in the processes, group communication) and the work lost by a worker when reconnecting to a new manager. Therefore, an analytical treatment of this problem would require a rather involved modeling of mechanisms at many different levels. Indeed, the simple approximation as represented by Equation (7.1) does not mirror the experimental results particularly well.

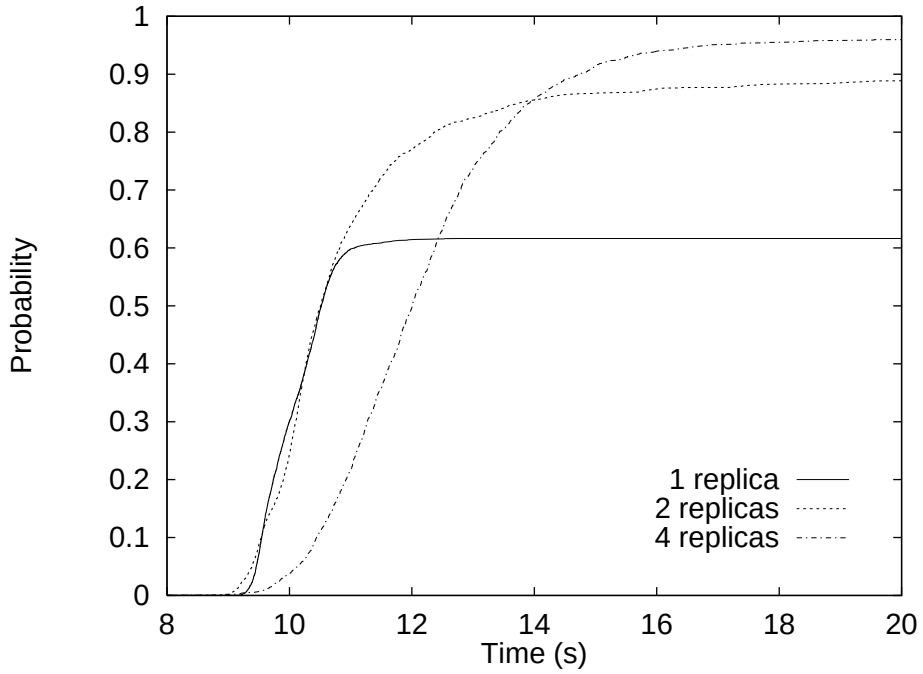

Figure 7.14: Runtime distribution of Calypso test program with varying number of masters, 50 ms granularity, faults injected with mean master lifetime 20 s, confidence bands narrower than 5%.

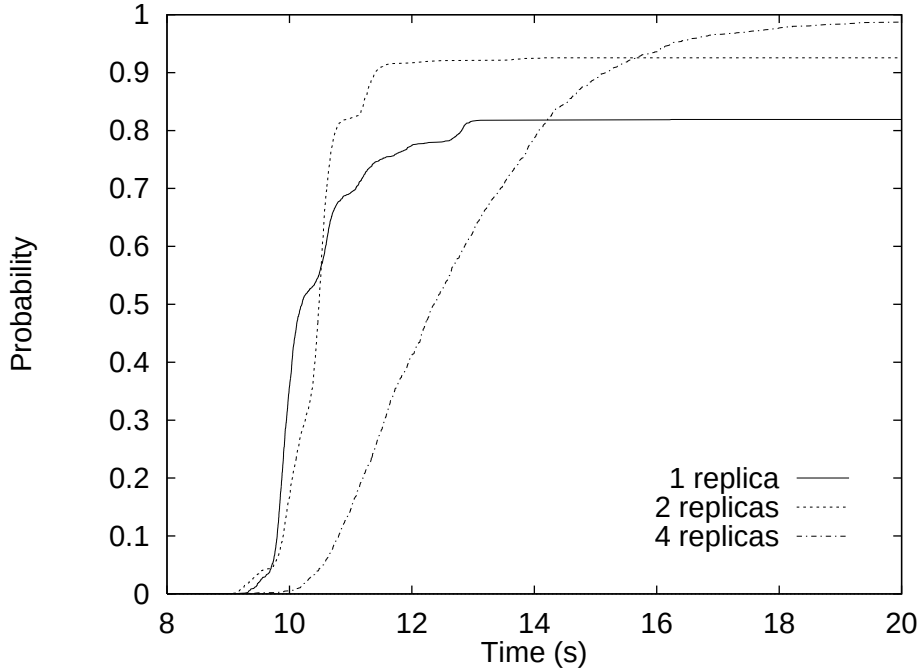

Figure 7.15: Runtime distribution of Calypso test program with varying number of masters, 50 ms granularity, faults injected with mean master lifetime 50 s, confidence bands narrower than 5%.

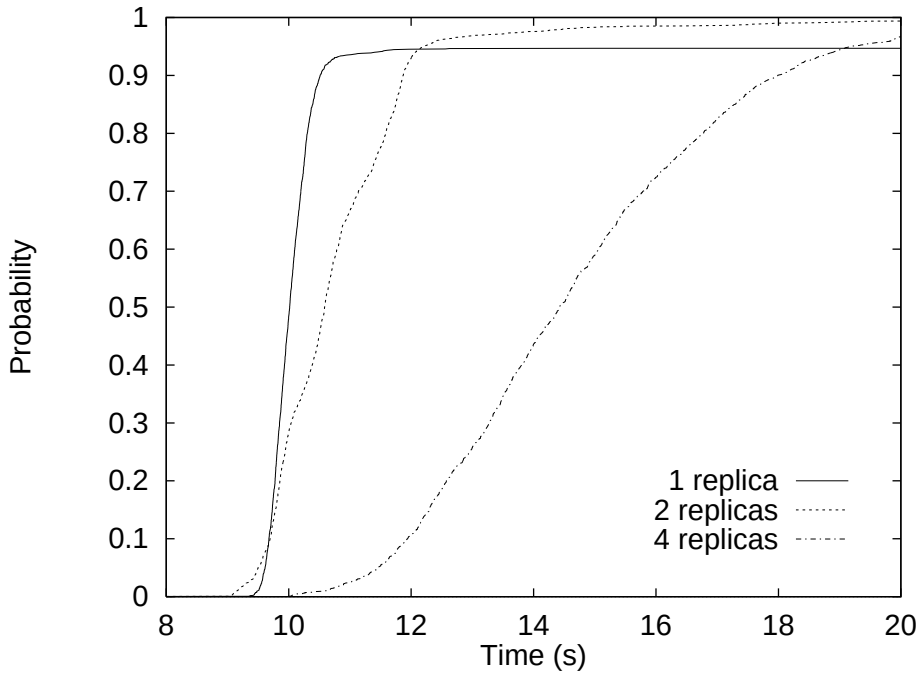

Figure 7.16: Runtime distribution of Calypso test program with varying number of masters, 50 ms granularity, faults injected with mean master lifetime 200 s, confidence bands narrower than 5%.

the SAS statistics software system).

Number of replicas	value	low	high
1	0.6162	0.6029	0.6293
2	0.8744	0.8627	0.8855
4	0.9395	0.9271	0.9503

Table 7.3: Responsiveness of Calypso program with varying number of replicated master processes at a deadline of 16 s and mean master lifetime of 20 s, columns show value estimate and lower and upper end of 95% confidence interval.

The case of mean lifetime 200 s shows interesting behavior: the responsiveness with four replicas is lower than with two or even one replicas. This lower value is easily explained by the large overhead of replicating four master processes. Moreover, with mean lifetime 200 s, the program with four replicas is slower (i.e., needs a longer time to reach a given responsiveness level) than at a higher fault rate. This behavior may seem counterintuitive at first, but is also explained by the larger overhead for more replicas: with respect to performance, it is actually beneficial if replicas die and thereby reduce the overhead, allowing the program to make faster progress. Such behavior reiterates the need to carefully choose the number of replicas for such a program.

7.4.4 Discussion

For the configuration used in these experiments, no performance benefits could be observed with replicated masters. However, replication does improve both the probability of eventually completing the program and its responsiveness. The high fault rates with which faults were injected in the experiments make a large number

Number of replicas	value	low	high
1	0.8182	0.8012	0.8343
2	0.9255	0.9120	0.9375
4	0.9355	0.9227	0.9467

Table 7.4: Responsiveness of Calypso program with varying number of replicated master processes at a deadline of 16 s and mean master lifetime of 50 s, columns show value estimate and lower and upper end of 95% confidence interval.

Number of replicas	value	low	high
1	0.9470	0.9349	0.9574
2	0.9848	0.9778	0.9900
4	0.7239	0.7019	0.7450

Table 7.5: Responsiveness of Calypso program with varying number of replicated master processes at a deadline of 16 s and mean master lifetime of 200 s, columns show value estimate and lower and upper end of 95% confidence interval.

of replicas necessary. Even with a moderate fault rate (200 s mean lifetime) four replicas have too high an overhead to be competitive for moderate deadlines, but remain superior with respect to the probability of eventually completing the program.

For a given deadline, similar tradeoffs can be observed as with checkpointing: For tight deadlines, low-overhead solutions (few replicas) can be preferable over high-overhead ones (many replicas) even though the former can have a lower probability of eventually completing the program. And also similar to the checkpointing experiments, the results of these experiments are rather specific to the given program and environment.

Unlike checkpointing, with replication there is always a chance of not completing the program at all, since only a finite amount of redundancy in space is available. Checkpointing will always eventually complete a program as it corresponds to an infinite amount of redundancy in time. A combination of both mechanisms appears useful: use replication to always service requests and checkpointing for basic fault tolerance.

7.4.5 Proposed improvements

There are a number of options to improve the performance and hence the responsiveness of replicated Calypso. Some have already been mentioned: masters can temporarily store results for parallel steps they have not yet reached (trading off time against memory), or they can shed load by closing worker connections if they detect that they lag behind other masters (also via the number of results for future steps). Another possibility to limit the divergence of masters is to introduce barrier synchronizations among the masters at the beginning of a parallel step. However, such a synchronization can slow down fast masters and the benefits would highly depend on the particular program and environment under consideration.

At the core of the performance problems of replicated Calypso is the group communication. Already the weakest of standard communication semantics, reliable delivery, is used. It is conceivable, however, that an aggressive group communication implementation with unreliable delivery of worker results could also be used: If it is ensured that all masters have at least one worker, they will make progress and eventually complete all parallel steps, even if they do not receive all results of other workers. Unreliable delivery reduces the overhead in the group communication but increases the complexity of the masters and the requirements on the master/worker association management. This approach requires further investigations.

7.5 Conclusions

In this chapter, the use of replication for increasing dependability and responsiveness has been considered. Analyzing the replication of a Calypso program has clarified the need to handle input/output data streams of legacy software in general in a flexible manner. The FT-DIO system addresses this need by providing a wrapper-based replication environment for any software that interacts with its environment over standard input/output streams. The particular strength of FT-DIO is its flexibility to adapt to different fault-tolerance requirements both at configuration and runtime and its easy extensibility.

FT-DIO uses the Totem group communication protocol to implement coordinated data exchange between replicas. The suitability of Totem for responsive systems is experimentally investigated and compared with analytical results. For simple fault scenarios, Totem has been found to indeed behave quite predictably. As long as the fault rates are not too high, and as long as the load on the participating systems can be controlled, Totem is a reasonable choice for a communication layer in a responsive system.

Based on FT-DIO, replicating the Calypso master could concentrate on the parallel step. While replication did not result in performance benefits in the experimental setup, the responsiveness of a Calypso program is indeed increased under heavy fault injection.

7.6 Possible extensions

Replication ensures practically uninterrupted service availability, as long as at least a single replica survives. However, in its pure form, replication does not prevent that eventually all replicas will experience errors and crash. Checkpointing, on the other hand, ensures that a service request will always make progress and eventually finish, but suffers from potentially long service interruptions. Hence, combining these two approaches could result in both the uninterrupted service availability (as given by replication) and the guarantee that a service will eventually finish (as given by checkpointing). It remains an open question how to integrate these two principles in a practical fashion (in particular with regard to legacy systems) and how to balance them (e.g., checkpointing interval versus replication level) so as to optimize end-to-end responsiveness. Experiments indicate that already very moderate degrees of replication should suffice; an observation that is corroborated by practical experience (e.g., the IBM Sysplex systems use a duplex design for high availability).

Chapter 8

Resource Guarantees for Parallel Programs

No program can be expected to complete by a given time if only insufficient resources are available. In a cluster of workstations that is shared with other users, programs by these users compete for resources and constitute a (possibly unpredictable) background load. In this chapter, a system is proposed that guarantees access to CPU time in a way suitable for parallel programs.

8.1 Introduction

The execution time of a program and hence its runtime distribution depend on the amount of resources available to it. Important resources are CPU share, memory, or I/O bandwidth (to disk or network). Multiple programs running on a single machine contend for these resources. Since Calypso targets settings where parallel programs can be run on machines that are also used interactively, a parallel program can have to contend for resources with interactive load (or other kinds of external load), too—it can be slowed down arbitrarily. This interactive load could be stochastically analyzed, a probabilistic model could be developed, and this model for background load could be taken into account when assessing the runtime distribution of the parallel program. This stochastic approach, however, has some disadvantages. On the one hand, the interactive user is not guaranteed a certain share of his own machine, with parallel programs using up resources in excess of what he might be willing to set aside for them. On the other hand, the stochastic model complicates the situation for the parallel program, since it in turn cannot rely on a minimum amount of resources on a single machine. It would be preferable for both interactive user and parallel program if they could rely on minimum resource shares.

This preference gives rise to the notion of a contract between (one or several) parallel applications and interactive, local users of a machine, regimenting how resources are allocated to each program and guaranteeing minimum and maximum share of these resources. Based on such a contract, the methods of Chapter 5 can be used to derive the runtime distribution of a Calypso program, where the relative speeds of each machine can express the available share of the actual machine. Therefore, a mechanism is needed that enforces such a contract and specifically considers the needs of parallel programs.

Enforcing such a contract could be done by the operating system, but a middleware approach (which does not modify a commodity operating system) is more in accordance with the COTS ideas. Several such middleware systems have been described; one of them is called “Scheduling Server” [226], a name that is here used summarily for all such systems. An implementation of such a scheduling server for the Linux operating system is presented in this chapter.

Most of the other, comparable systems are concerned with non-parallel applications running on only a single machine. The case of parallel applications executed in a distributed fashion on a cluster of workstations is more challenging. One aspect is the programming and usage patterns of the parallel program. For a Calypso program, it is reasonable to assume that the master process can make use of a machine that is (in some sense) dedicated to the parallel program and not subjected to resource sharing—this is even more likely since the master is often not a major source of resource consumption. Hence, a Calypso program can be used

asymmetrically with workers running under resource control, and the master having unrestricted access to a single machine. But this is not necessarily the case for all kinds of parallel programs. In a true BSP-like programming model, processes are symmetric and there is no reason to consider one process specifically. Hence, there are two cases to consider for parallel programs running under resource control: one were all processes are symmetric and synchronize with each other, the other one were all processes only communicate with one special process.[1] The first case is evaluated with a true BSP program, the second case with a Calypso program.

As experiments show (see Section 8.4), a naïve use of distributed scheduling servers results in unacceptable performance for symmetric programs. This is commensurate with the well known fact that temporally coordinating the execution of distributed program parts can have an immense effect on parallel program efficiency. The scheduling server design presented here combines both aspects: Enforcing a resource contract while at the same time providing coordinated execution of parallel programs running under this contract. The work described in this chapter is partly based on initial ideas first presented in [135] (in joint work with A. Polze and M. Werner); the solution presented here is described with more supporting experiments in [132, 133].

The rest of this chapter is organized as follows. Related work is described in more detail in Section 8.2. In Section 8.3, limitations imposed by commodity operating systems are considered and the design of a the scheduling server is explained. Some experimental results are given in Section 8.4, conclusions are presented in Section 8.5, and possibilities for extending this work are outlined in Section 8.6.

8.2 Related work

8.2.1 Predicting or controlling CPU share

As discussed in the introduction, resource availability, here in particular CPU share, can be either predicted or deliberately controlled. Examples for the predictive approach for network resources are the Network Weather Service [307], which uses distributed probes to actively and passively measure CPU and network parameters and a set of stochastic prediction techniques, or similarly the Network Status Predictor [146], which works with amazingly simple stochastic methods. Examples for CPU resources are the estimation of slowdown of workstations when programs have to contend for resources (and allocation decisions based on this estimation) [78] or using normal distribution-based performance models to obtain intervals for the likely behavior of a program [250].

The alternative is to control (and not only predict) the CPU share allocated to a given program. This is a straightforward task in a real-time environment such as, e.g., real-time Mach; LEE et al. [168] report experiences with such an approach called processor reservations in real-time Mach. An interesting example for CPU scheduling with real-time algorithms in a commodity operating system is described by DENG et al. [66]: They replace the Windows NT operating system scheduler with a two-level hierarchical scheduler that allows the coexistence of non-real-time and real-time applications in an open system; real-time applications are started if an acceptance test determines that sufficient resources are available (it is not necessary to perform a scheduling analysis for the entire set of applications); and real-time applications can choose from a number of different scheduling strategies, which are then implemented independently from other real-time applications running at the same time.

Other systems for CPU share control target commodity off-the-shelf operating systems and try not to modify them. POLZE [226] describes a Scheduling Server for NeXTSTEP. This server is a program that runs with the highest available "real-time"[2] scheduling priority and cyclically suspends or resumes a controlled program. The controlled program also runs with a high "real-time" scheduling priority, higher than normal programs, but strictly lower than the scheduling server's. Since the scheduling server itself is suspended practically all the time and only wakes up to schedule the controlled program, it does not use too much

[1] From the perspective of resource management, a Calypso master process running under control of a scheduling server is equivalent to a BSP program.

[2] "real-time" here refers to fixed priority scheduling and is a technical term of NeXTSTEP.

time and does not interfere with other programs. The resource share given to the program is tunable via the scheduling server. This approach shows excellent stability of resource allocation even under considerable background load (see [239] for experiments).

Similarly, the URSched system [125] uses the fixed-priority scheduling of Sun's Solaris 2.4 operating system to provide smooth video playback, considerably reducing jitter compared to the standard time-sharing scheduling strategies. The implementation technique is practically identical to the Scheduling Server described in [226]. In [53], an extension of URSched is described that implements rate monotonic scheduling, admission tests, and fairness properties; a later paper [180] describes an implementation of URSched for Windows NT.

A comparison between user-level approaches (like URSched and the Scheduling Server) and operating system approaches (e.g., processor reservations) shows that for the price of modifications to the system kernel, more precise accounting of resource usage (in particular, processing time spent in the operating system on behalf of a process) can be achieved. User-level approaches on the other hand allow better flexibility to implement various advanced scheduling schemes on top of existing operating systems (as demonstrated by [53]). Also, there are various possibilities what to do with a controlled process during its inactive period: it can be suspended or set to a low time-sharing priority.

8.2.2 Coordinated scheduling

The question of temporal coordination of processes belonging to a parallel program has been addressed by work on gang scheduling and coscheduling. While there is no uniform usage of these terms in the literature, and additionally the usage has changed over time, the general consensus today (and the interpretation adopted here) seems to be that gang scheduling refers to centralized control schemes and explicit information, whereas coscheduling tries to identify and use information implicit in the program execution to achieve temporally coordinated execution of program parts executing in parallel.

The original idea of gang scheduling is to schedule distributed threads of a parallel program, running on multiple processors, at the same time [219]. Such coordinated scheduling provides these threads with an environment similar to a dedicated machine and allows them to spin-wait for synchronization and/or communication. Spin-waiting is particularly important for fine-grained programs that would suffer considerably from the context switches entailed by blocking communication. On the other hand, for coarse-grained or highly imbalanced threads, blocking can be beneficial.

FEITELSON and RUDOLPH [76] give a characterization of situations in which either gang scheduling or blocking is preferable. They do so by analyzing BSP-style programs with a simplified system model. While such simplifications are appropriate for dedicated multiprocessor systems, the higher complexity of networked workstations (e.g., their time-sharing schedulers or potential network collisions) does not allow such an approach. The experimental results in [76] are also closely tied to custom-made hardware. FRANKE et al. [83] describe SHARE, a newer gang scheduler for the IBM SP-2. The design of SHARE introduces a hierarchical control scheme for gang management, however, the anticipated scheduling periods are on the order of several minutes, making this approach unacceptable for use in workstations that are potentially used interactively (this long period is due to the high overheads of switching application contexts on the SP-2). SHARE is also unique in handling children of controlled processes correctly.

Classic gang scheduling research assumes that the parallel machine is dedicated to parallel programs, which basically use it in a time-sharing fashion. Dedicated use can usually no longer be assumed if clusters of workstations are considered. In particular, pure spin-waiting is no longer acceptable since it is a waste of resources. DUSSEAU et al. [73] and later ARPACI-DUSSEAU et al. [14] suggest to use adaptive two-phase spin-blocking communication that, in combination with a standard operating system's time-sharing scheduler,[3] results in coordinated scheduling of distributed threads. In this scheme, a communication event (sending or receiving of messages) is considered as an implicit request for coscheduling between sender and receiver. They show that this implicit coscheduling is particularly beneficial for fine-grained programs. The essential fact exploited here is the scheduler's property to give a priority boost to a blocked program after becoming

[3] More precisely, the scheduler of UNIX SVR4 as implemented in Sun's Solaris operating system.

runnable again. SOBALVARRO et al. [265] argue along similar lines, but use additional hardware support (a programmable network interface) and system support (a custom-made network driver) to generate additional scheduling events when messages arrive. The idea of coscheduling has also been applied to proportional-share scheduling [13] by increasing the likelihood of scheduling sleeping processes (mirroring the SVR4 scheduler's behavior). While this work does combine implicit coscheduling and fairness for parallel applications, it is necessary to modify the standard operating system to do so and for soft real-time applications, an algorithm with high overhead is necessary.

HORI et al. [108] give a performance comparison of gang scheduling and coscheduling for data-parallel workloads on a PC cluster. They have implemented gang scheduling via a signal-based distributed user-level scheduler, SCore-D, which is quite similar to the design presented here. Unlike the system describe here, SCore-D can assume complete control over scheduling and has therefore not to take any precautions against background load; consequently, it does not use fixed-priority scheduling and is not concerned with performance stability under background load. Also, they could make use of advanced network technology (Myrinet), whereas the experiments reported here were done on traditional 10BaseT Ethernet.

8.3 Prototype description

There are two main aspects to the prototype system: First controlling CPU resources with a scheduling server on a single machine, second synchronizing distributed scheduling servers.

8.3.1 Controlling CPU share

The prototype was developed for the Linux operating system (using kernel version 2.0.31). Newer Linux kernels provide fixed-priority scheduling classes that are not subjected to aging (similar to the schedulers of, e.g., Solaris or NeXTSTEP). Therefore, the basic structure is the same as in [226] or [125]: A process with highest possible fixed priority acts upon controlled processes, which run at a medium fixed priority, in a cyclic fashion, periodically making time slices available to the controlled process. This sequence of time slices forms the schedule for the scheduling server: for each slice, there is information if it is free or owned by a given process.

There are two basic possibilities for acting out scheduling decisions at the beginning and end of a time slice: one is to manipulate scheduling priorities, the other is to explicitly suspend/resume the controlled process that owns it. The natural implementation for the second alternative under Linux is to send SIGSTOP and SIGCONT signals (see Figure 8.1). This signal-based approach is used since it showed to have better stability under background load compared to an implementation that manipulates process priorities (see Section 8.4.1) and also gives clearer semantics as to how much resources are actually allocated.[4] Typical time slices are 10 ms or 20 ms, owing to the timer tick resolution in Linux (and many other COTS operating systems) and to the need to accommodate interactive users. The use of standard signals should make this prototype easily portable to any operating system that supports fixed-priority scheduling classes.

8.3.2 Synchronizing distributed schedulers

Work on gang scheduling and coscheduling has conclusively shown the need to temporally coordinate the execution of distributed processes of a parallel application to obtain acceptable performance. Since the scheduling server controls the times of execution of these processes, it follows that the distributed servers need to synchronize their scheduling activities.

Consider a simple example of an application consisting of three processes running on three separate machines, each running with a single time slice per cycle. If the time slices are not synchronized, there is no guarantee that they coincide in time (as sketched in Figure 8.2); therefore, a simple round-trip message between two processes on different machines would take an entire scheduling cycle (assuming the receiver can

[4] Priority-based implementations do not give clear upper bounds on CPU share, only lower bounds—this is somewhat related to the difference between constant utilization servers [65] and total bandwidth servers [268] in real-time scheduling.

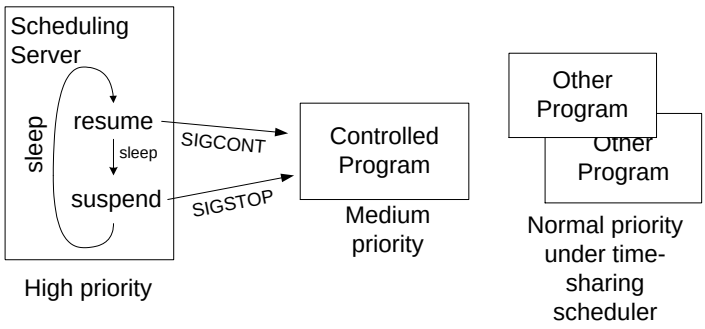

Figure 8.1: Structure of a signal-based scheduling server.

answer immediately and blocking communication primitives are used; see an extended discussion in [135]). In a synchronized case, such a message would only need twice the network delay plus protocol processing costs (with spinning receive, to avoid descheduling a process due to blocking on a receive).

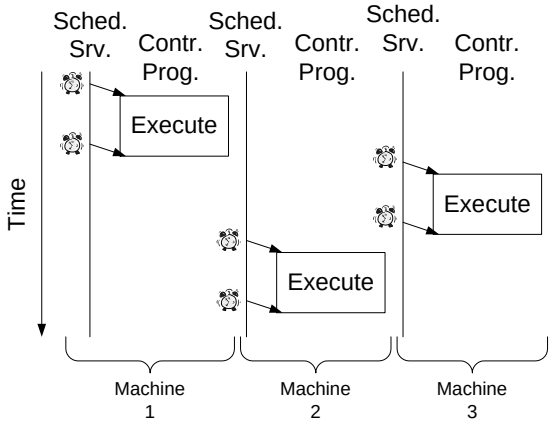

Figure 8.2: Unsynchronized scheduling servers with a single controlled program, distributed over three machines.

Ideally, synchronizing the scheduling servers should be done in a decentralized fashion, with only a few and rare control messages. An obvious way to achieve this is to use one of the well known clock synchronization protocols (e.g., [61, 158]). The basic idea is for a scheduling server to send its remaining sleep time to its peer servers, which then sleep for this amount of time, too (after adjusting for network delay). The problem with this approach is that it requires timer interrupts of well under a millisecond granularity to achieve acceptable synchronization. However, the Linux operating system only provides timer interrupts with 10 ms granularity.[5] This granularity would only allow long scheduling periods and make the scheduling server unsuitable for interactive use.

Therefore, there is a need for generating interrupts that lead to execution of the operating system scheduler at more or less arbitrary points of time. Message arrivals do generate interrupts. Hence, messages to the scheduling servers can be used to start their execution. In the initial prototype, this fact is exploited by designating one of the scheduling servers as master that sends a short control message (one byte) to the slave scheduling servers, causing them to wake up (see Figure 8.3).[6] This control message only contains the current slot number in the schedule; the slave servers then suspend, if necessary, the process occupying the previous slot and resume the local process that corresponds to the current slot. The actual schedule only has to be communicated between machines when a parallel application starts or finishes. Since these slave servers run

[5] With the `setitimer` function, measured on a Pentium 166 MHz.

[6] Currently, this is implemented via point-to-point messages. Reliable broadcast messages could also be used.

at highest fixed priority, but are usually blocked in the receive call for this control message, they will be scheduled immediately after message arrival. In this design, only the master server sleeps and wakes up in a time-triggered way; the scheduling activities of the slave servers are driven by message arrival from the master server (hence the name "message-driven" for this concept in [135]).

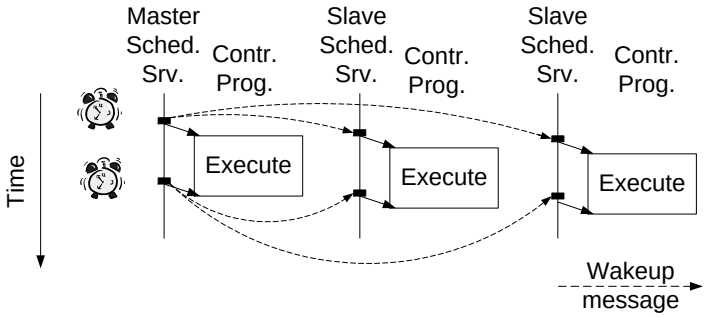

Figure 8.3: Message-driven synchronization of scheduling servers.

This results in coordinated resuming of the controlled program's distributed processes. There is a certain gap between the actual times the processes start, caused by the network delay and the processing delay in the end systems. This gap can be inconvenient, but it is not necessarily detrimental: To improve performance of BSP programs, DONALDSON et al. [68] suggest a random backoff approach for distributed processes after a barrier communication to avoid collision on a shared medium, similar to the backoff mechanism used in Ethernet. The message-driven scheduling servers produce a similar effect by the slight offset it takes to resume the processes.

Note that this does not mean that messages between the distributed parts of the application have to pass through the scheduling servers. Indeed, the application itself can completely ignore the fact that it runs under control of such a distributed scheduling server; there is no need to modify it in any way. It is also conceivable to include an API such that an application can receive and provide feedback from and to a scheduling server.

With a single master scheduling server, this system is not fault-tolerant since a failure of the master would result in the slave servers never receiving trigger messages again. It is simple to protect against it: All the slave servers know the schedule and can therefore compute an upper bound on the remaining sleep time. They can hence arrange for a timeout to occur at the earliest timer interrupt after this sleep time. If the master server works correctly, it will wake up all the slaves before this timeout expires, which can then reset the timeout. If the master fails, the slaves will wake up out of synchronization, but independently. The timeout limits the delay caused by a failed master to the timer granularity, which is often 10 ms, depending on the operating system.[7] The first slave to wake up without a master message installs itself as the new master and wakes up its former peers; the old master is assumed to have failed. If two or more slaves attempt to become a new master, ties can be broken arbitrarily, e.g., following some identifying number of the servers.

8.4 Some experiments

For an experimental evaluation of the proposed prototype, four Pentium 90-based PCs, running Linux version 2.0.31 (completely unmodified) and connected by standard 10 Mbps Ethernet were used. The machines were in exclusive use, but some external network traffic during the experiments could not be completely ruled out. Both symmetric BSP-style programs and asymmetric Calypso programs (in the sense of Section 8.1) were considered; the parameters for both programs follow the description in Section 4.3.

The results for the stability of the scheduling server under background load on a stand-alone machine are shown in Section 8.4.1. In Section 8.4.2, the behavior of a BSP-style program under scheduling server control

[7] Assuming that there are no other processes running at the same or higher fixed priority are present in the system.

is discussed, and Calypso programs are investigated in Section 8.4.3. Only a few cases are discussed here, a more detailed exposition of experimental results can be found in [132, 133].

8.4.1 Stability

The main purpose of the proposed scheduling server is to allocate a predetermined share of the CPU to a controlled program and also to make sure that this program does not exceed its share. To investigate the stability of this allocation under increasing load on a stand-alone machine, a slightly modified version of Linpack [69] was used as controlled program. This version computes the actual floating point performance received during its runtime; this number can be interpreted as the relative share of CPU time (similar to [239]).

In Figure 8.4, Linpack's performance under scheduling server control is shown, with 50 runs done for 0, 1, . . . , 10 computation-bound background processes each,[8] and reserved CPU share ranging from 10% to 90%. While the behavior is far from perfect, the CPU share allocated to Linpack is reasonably stable: For all combinations of CPU share and background load, the variation coefficient of the performance delivered to the application is less than 4 %.

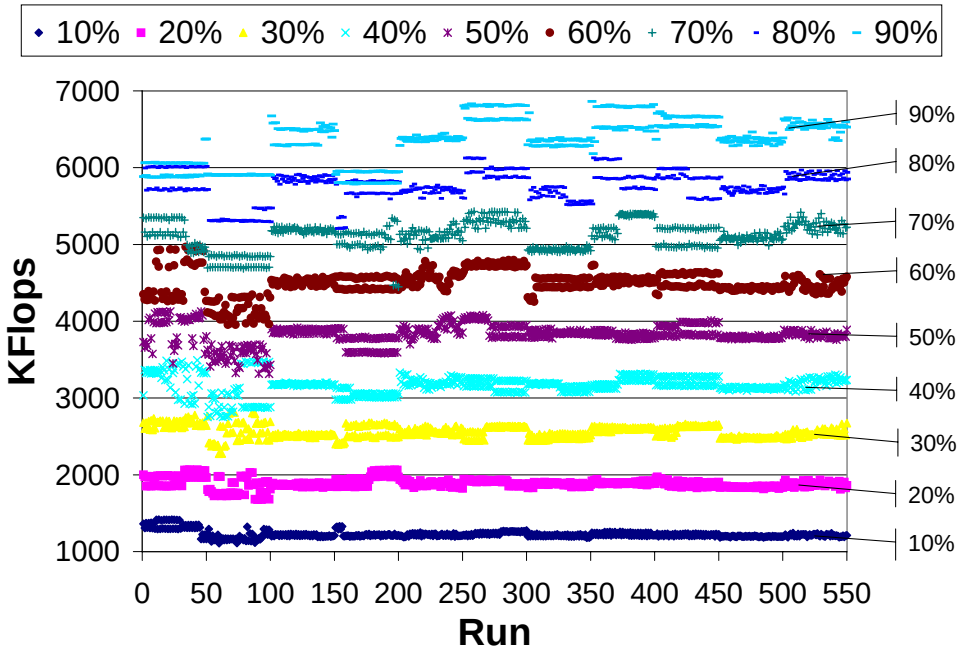

Figure 8.4: Linpack under scheduling server control: received CPU share (in KFlops) for successive experiment runs, background load increases every 50 runs, shown for various amounts of reserved CPU share ($10\%, \ldots, 90\%$).

The reserved share of CPU time should not exceed about 50%, since at least the operating system itself needs time for its own operation. Stability degrades considerably beyond this point (a priority-based implementation behaves similarly).

8.4.2 BSP programs and scheduling servers

The basic building block of a BSP program is the barrier synchronization. The following Figure 8.5 gives the times in milliseconds for a single barrier synchronization, without scheduling server control and using usual blocking communication primitives. In this as in all following figures, the granularity g is shown in milliseconds and ranges from 0.1 ms to 100 ms, covering fine-grained to large-grained applications. Load imbalances v investigated in these experiments are 0%, 25%, 50%, 75% and 100% (the five bars shown for

[8] More precisely, runs $50k$ to $50k + 49$ were done with k background processes, k ranging from 0 to 10.

each granularity).[9] Times shown are averaged over 50 runs.

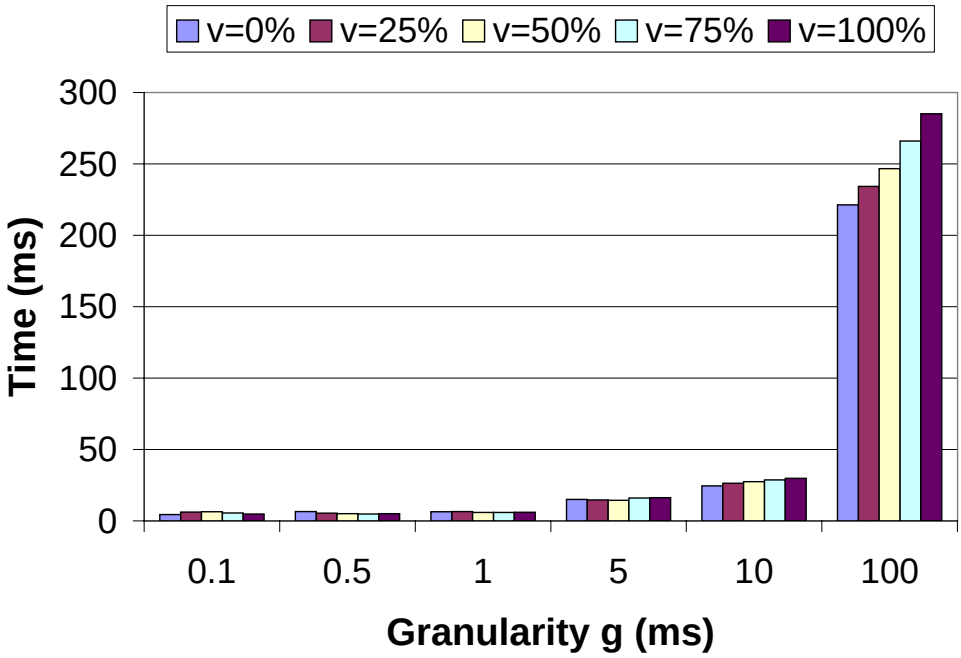

Figure 8.5: Average runtime of a single barrier synchronization without scheduling server, blocking communication, shown for various granularities g and load imbalances v.

Using unsynchronized scheduling servers on all machines results in the numbers shown in Figure 8.6. Here as in all other experiments, the scheduling server allocated 2 out of 10 time slices to the controlled program, where one time slice is 20 ms. Evidently, with unsynchronized scheduling servers, the slowdown is much larger than the expected factor of five—an entire scheduling round is necessary to complete a single synchronization. While this mode of use has very good predictability, the performance is unacceptable for practically all applications.

The effects of the synchronization mechanism for scheduling servers proposed in this chapter are shown in Figure 8.7. Again, these times should be at most five times as long as in Figure 8.5, but they are actually better. For large-grained programs, the slowdown approaches roughly 4.5. For fine-grained programs, it is only about 1.5 to 2; since the program is communication-bound and not computation-bound, the CPU limitation is not as severe. This smaller than expected slowdown can be attributed to both coscheduling effects and to the higher priority used by the processes.

A typical BSP program consists of more than one barrier synchronization, and also includes some communication. As an example for such a program, with 50 barrier synchronizations and all routines exchanging data with all others after each synchronization, Figure 8.8 shows the runtime without scheduling server, Figure 8.9 with synchronized scheduling servers. For such heavily communicating programs, the use of spin-blocking communication primitives (here 200 μs spin time is used) turns out to be advantageous; a full discussion can be found in [132]. Again, using unsynchronized scheduling servers yields unsatisfying results.

These experiments (along with the extended set of experiments in [132]) indicate that for symmetric, synchronizing parallel programs, e.g., those of the BSP programming model, synchronizing scheduling servers with the message-driven approach proposed here is both necessary and feasible. KARL [132] also discusses the impact of background load and shows that parallel programs do receive a predictable CPU share even with varying levels of background load.

[9] As defined in Section 4.3, a load imbalance v for a granularity g means that the runtime of a routine is randomly drawn from a uniform distribution $\mathcal{U}(g-(g*v)/2, g+(g*v)/2)$.

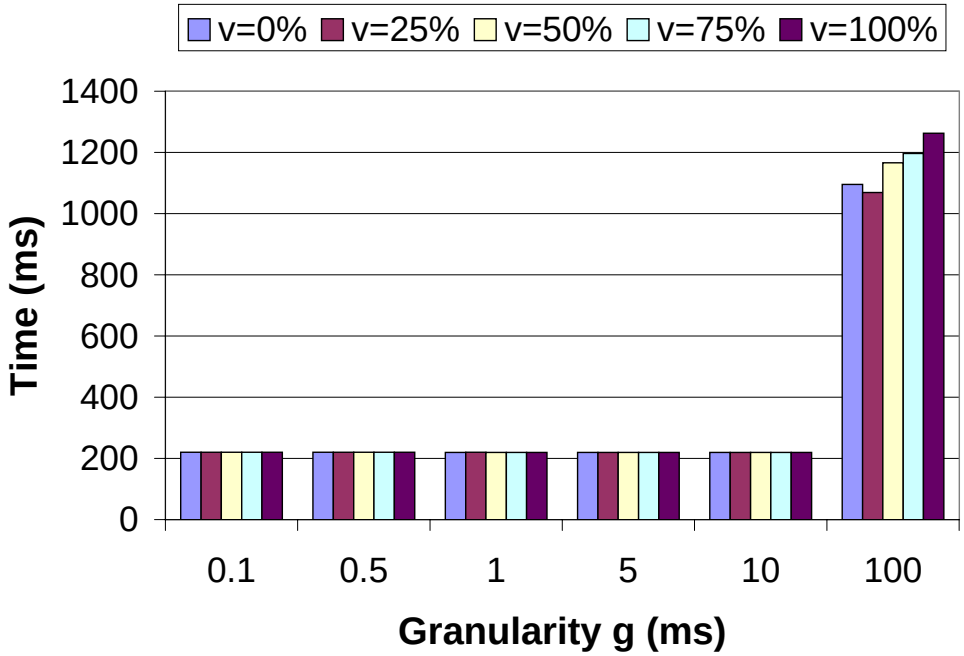

Figure 8.6: Average runtime of a single barrier synchronization with unsynchronized scheduling servers, blocking communication, shown for various granularities g and load imbalances v.

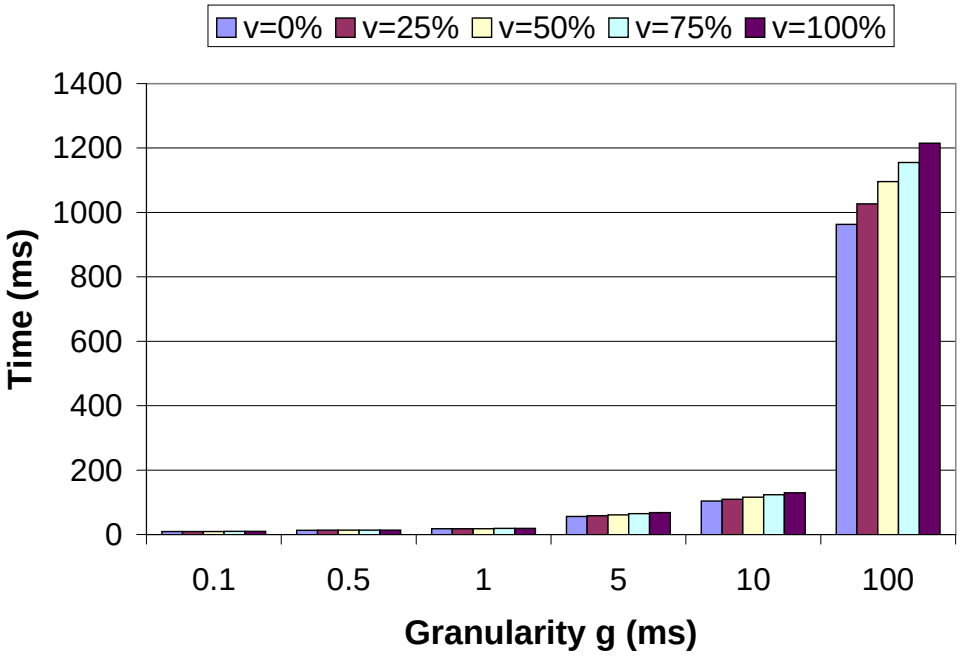

Figure 8.7: Average runtime of a single barrier synchronization with synchronized scheduling servers, blocking communication, shown for various granularities g and load imbalances v.

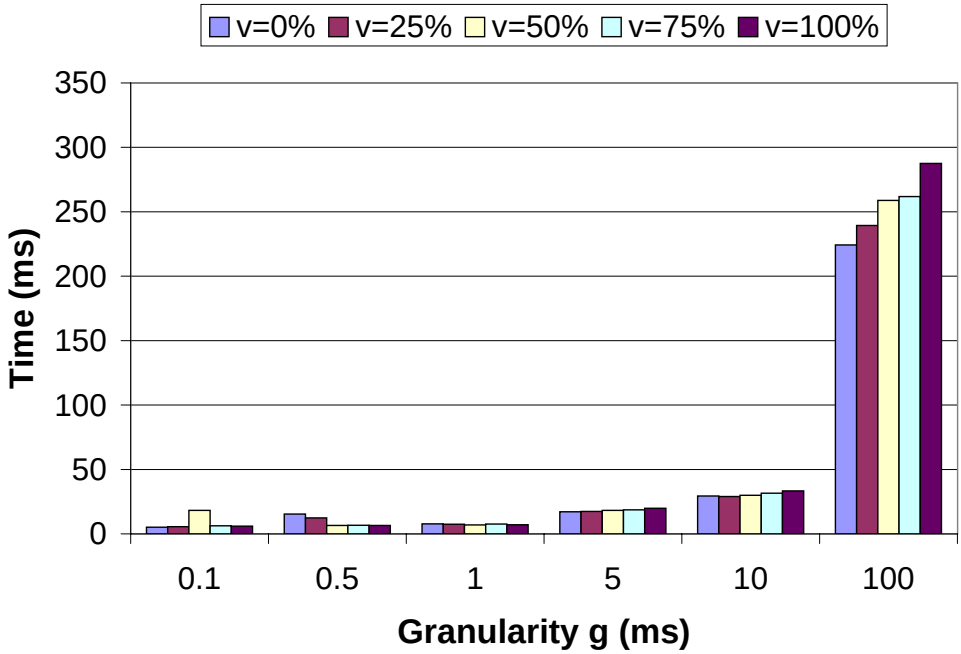

Figure 8.8: Average runtime of a BSP program with complete communication pattern, 50 synchronizations, spin-blocking communication (200 μs), no scheduling server, shown for various granularities g and load imbalances v.

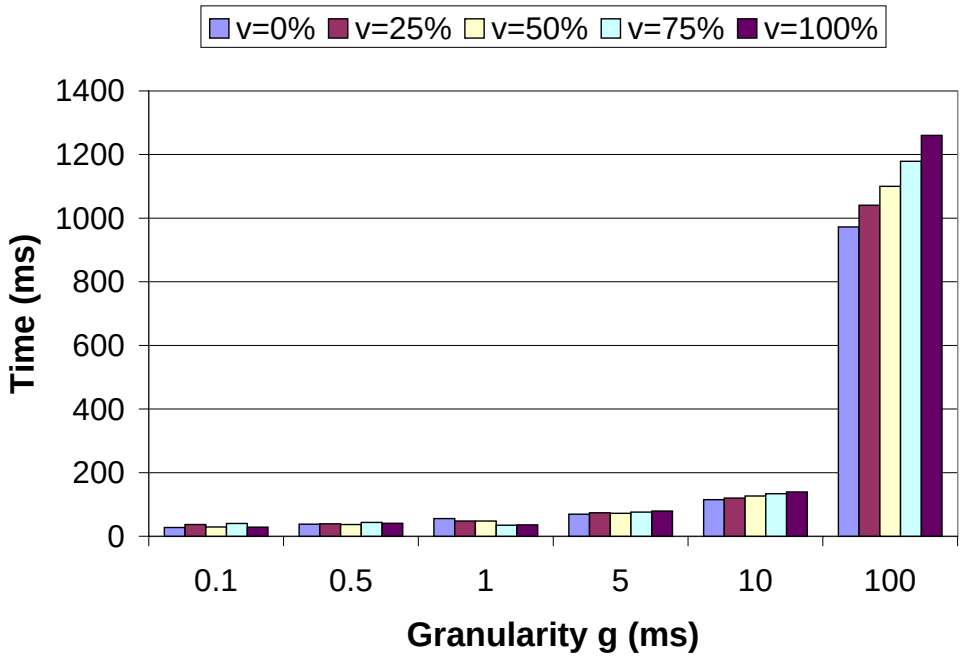

Figure 8.9: Average runtime of a BSP program with complete communication graph, 50 synchronizations, spin-blocking communication (200 μs), with synchronized scheduling servers, shown for various granularities g and load imbalances v.

8.4.3 Calypso programs and scheduling servers

In this section, the use of Calypso programs is considered with the Calypso master process not subject to a scheduling server: the master is always eligible to run, but (typically) shares a machine with a worker process.[10] If the Calypso master is also eligible to run only periodically, the same line of argument as in Section 8.4.2 applies and synchronizing these executions is necessary.

While Calypso uses a BSP programming model, its actual implementation follows a master/worker style, enhanced with load balancing techniques. There is no explicit synchronization of all processes (either worker or master) in a Calypso program. Hence, the main motivation for synchronizing the processes with each other is missing. Moreover, if worker processes are synchronized with each other, they all attempt to access the master at (potentially) the same time. This can cause the master and the network to become a bottleneck, in particular for programs with high traffic, and suggests different behavior of Calypso programs than BSP programs when run under scheduling servers.

The Calypso program from Section 4.3 is used for experiments. As that section has indicated, the imbalance parameter v is of lesser importance, whereas the previous discussion suggested a potential impact of the traffic parameter a.[11] Therefore, the experiments used granularity g and traffic a as parameters, the scheduling servers were again set to provide time slices of 20 ms every 100 ms, spin-blocking with 200 μs was used, and the numbers shown here are averaged over at least 50 runs.

Figure 8.10 shows the results with unsynchronized scheduling servers, Figure 8.11 with synchronized scheduling servers; for easier comparison, the ratio of times with synchronized servers divided by times with unsynchronized servers is provided in Figure 8.12. The scenario here is much more complicated than in the case of BSP programs. The overall behavior is such that for most cases, synchronization actually harms the performance, but with increasing granularity, synchronization becomes competitive and outperforms unsynchronized servers. This is commensurate with the initial discussion: with larger granularity, fewer requests are made at the master process, which becomes less of a bottleneck. However, the exact point where synchronization becomes beneficial is highly dependent on the actual program, and differences are not very big anyway. Synchronizing scheduling servers has the additional advantage that in almost all cases the variation coefficient of runtimes is smaller.

Only numbers for a balanced load are shown here. For other settings of the imbalance parameter, the behavior is similar, however, the effects of load balancing are reduced by the slotted CPU availability under scheduling server control. Therefore, programs with unbalanced load perform comparably worse under scheduling servers than they do without.

8.5 Conclusions

In this chapter, the problem of controlling the CPU share given to a parallel program running on a cluster of workstations and the effects of such a control on the distributed execution of a parallel program have been considered. A prototypical implementation of a scheduling server for the Linux operating system has been presented. Among several design choices, a signal-based implementation has been used due to its stability and portability.

While such a straightforward implementation works well for stand-alone programs, the performance impact on symmetric parallel programs (as represented by the BSP programming model) proved to be disastrous. A synchronization mechanism has been suggested and implemented that copes with the limited clock resolution of PC-based commodity systems and still achieves reasonable performance for symmetric parallel programs, even in the presence of background load. The performance benefits of this synchronization mechanism are due to the fact that coscheduling is achieved by this synchronization mechanism. For asymmetric parallel

[10] Since the worker process is subjected to scheduling server control, the master process runs at a fixed priority higher than the worker, but lower than the scheduling server—otherwise the master would not service any worker requests while the local worker is running.

[11] Following the description of the test program in Section 4.3, a traffic parameter a indicates the number of pages of size 4 KBytes that are read and written by every routine.

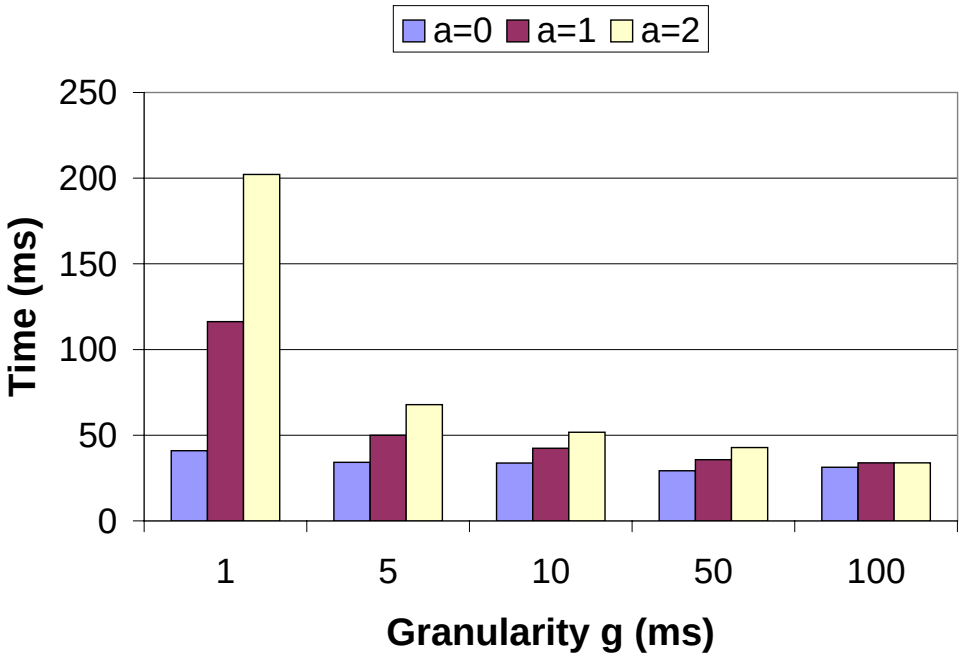

Figure 8.10: Average runtime of a Calypso program with unsynchronized scheduling servers, shown for various granularities g and traffic parameters a.

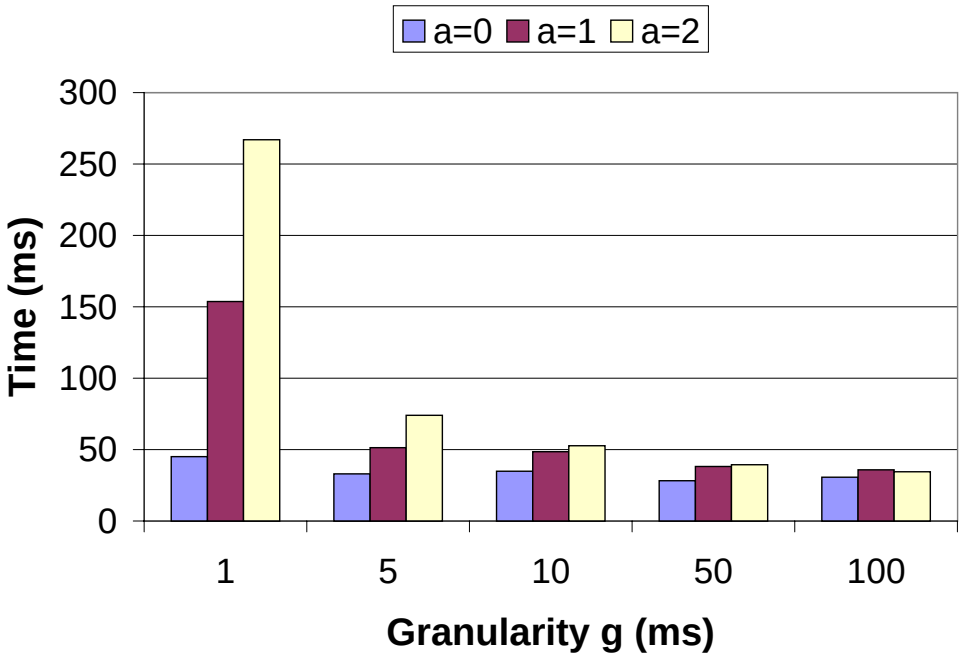

Figure 8.11: Average runtime of a Calypso program with synchronized scheduling servers, shown for various granularities g and traffic parameters a.

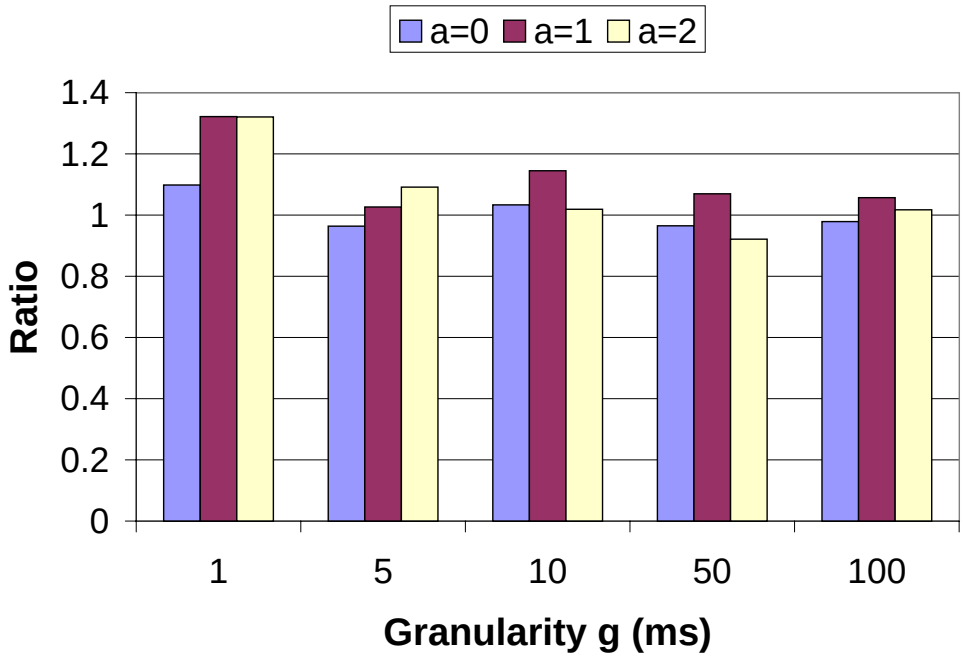

Figure 8.12: Ratio of runtimes of a Calypso program, comparing synchronized and unsynchronized scheduling servers (larger values indicate that unsynchronized scheduling servers perform better), shown for various granularities g and traffic parameters a.

programs (like Calypso programs), synchronization is superfluous in many cases and can, in particular for very traffic intensive programs, actually harm performance.

Using extensive measurements, the influence of various parameters like granularity, load imbalance, or communication pattern were investigated. There are several observations for symmetric programs, the most relevant ones are: for fine-grained, moderately communicating programs synchronized scheduling servers provide a reasonable means of achieving coscheduling; for heavily communicating programs, spin-blocking has to be added; synchronized scheduling servers reduce the variation coefficient of program execution time. For asymmetric parallel programs, coscheduling can actually be harmful since the master process can become a bottleneck.

8.6 Possible extensions

There are a number of possibilities to extend this work. For the experiments described in Section 8.4, only four PCs were available. This small number does not allow to address questions of scalability. To do so, experiments with a larger cluster, along with a reimplementation of the synchronization to use multicasting (or a tree-based dissemination of the synchronization messages if no multicasting is available, similar to Score-D), are desirable. Some more experiments with multiple distributed programs running under scheduling server control at the same time or with additional, uncontrolled background load could also be performed.

A more generalized version of a scheduling server should not only control CPU resources, but other resources as well. Memory is an obvious candidate; Linux implements the necessary memory locking primitives to make this relatively straightforward. Networking bandwidth is more complicated and depends on what kind of network is to be used. In ATM, e.g., a priori reservation of bandwidth is a possibility but should be integrated with Quality of Service end system architectures.

As has been discussed in Section 8.3.2, the coarse resolution of operating system timers is a major obstacle for an efficient, time-driven synchronization of distributed servers. A promising opportunity is represented by the UTIME extensions [21] to the Linux kernel that promise programmable timers with microsecond accuracy.

However, it remains to be seen how this timer accuracy can be provided to application processes as well (and not only kernel modules).

To make such a distributed scheduling control practical, it has to be integrated in a general resource management scheme. BARATLOO et al. [24] propose a resource management system that fits particularly well with the prototype described here. The distributed scheduling server can be used to act out Quality-of-Service related decisions made by the resource broker. Programs that fork off remote processes could at the same time inform the resource broker that they require gang scheduling for this process. Some additional effort for integrating these two systems as well as some studies regarding Quality-of-Service specifications and policies are needed here; the Globus resource description language [89] could serve as a starting point. Additionally, such an integrated resource management/scheduling system can react flexibly to the requirements of tunable applications [49].

Chapter 9

Reaching out to Wide Area Networks

Satisfying the unlimited resource needs of parallel applications leads to the prospect of metacomputing in geographically widely distributed environments. In such wide area environments, communication can become a serious bottleneck. To reduce communication overhead, an annotation scheme for communication patterns of a program is proposed here. This scheme is investigated with an implementation in the Charlotte system. It is shown that these annotations considerably increase Charlotte's efficiency and can also serve as a stepping stone for responsive metacomputing. Additionally, an infrastructure for resource allocation in wide area networks is briefly discussed.

9.1 An opportunity and a challenge of Wide Area Networks: metacomputing

For a distributed program, the communication can have a large impact on the program's performance. It therefore appears reasonable to attempt to limit the amount of data that has to be transmitted between, e.g., master and workers in a Calypso-style parallel programs. However, in a cluster environment, the runtime overhead necessary for administrating the necessary state information outbalances the time saved by reducing network traffic (as shown by experiments with Calypso).

But with the increasing availability of Wide Area Network (WAN) connectivity via the Internet and its growing capacity, interest in computing in WAN environments has been ever increasing—and here, the trade-offs between local computation and communication should be different. Computing in such wide area networks has often been called metacomputing [96, 263].

Metacomputing over the Internet is a tempting opportunity for parallel and distributed computing. The Internet can be regarded as a potential source for an immense computational capacity. A vast number of often idle machines are connected and provide—in principle—enormous resources to tackle large problems. Resources can be in the form of individual or clustered workstations or even cooperating large supercomputers. Reaching out to these resources allows to handle a new dimension of problems unapproachable by purely local means.

But metacomputing also poses its own challenges that so far have prevented this vision from becoming reality. Among these obstacles are heterogeneity, security concerns, the need to install programs on remote computers, high communication latencies and low bandwidth, and the inherent unreliability of remote machines and the Internet itself. Predictability of program execution is very difficult to achieve, given the lack of control over remote resources and the missing Quality-of-Service guarantees of today's Internet technology. While there are efforts to ameliorate this situation (e.g., RSVP [42] or Differentiated Services [36]), it is at the moment unclear how successful they will be.

The Charlotte system [27], a close relative to Calypso, is an attempt to overcome some of these obstacles. In this chapter, the focus is one the problem of communication overhead in WAN environments. Charlotte serves as a test case for the concept of reducing communication overhead by program annotations. In Section 9.2, an extension to Charlotte is described that implements such an annotation scheme for the communication pattern of a Charlotte program; additionally, these annotation provide a basis to argue about the

resource requirements of a Charlotte program. This knowledge is also necessary for responsive execution of a Charlotte program as soon as Quality-of-Service guarantees are available in the Internet. More details about this extension can be found in [130].

Additionally, finding suitable resources for program execution is much more difficult in a WAN than in a LAN. Section 9.3 contains a description of KnittingFactory, an infrastructure to enable distributed, volunteer-based resource allocation and to support wide area computations in general. KnittingFactory was developed in joint work with A. Baratloo, M. Karaul, and Z. Kedem; an extended discussion can be found in [25, 26]. Finally, the chapter is concluded in Section 9.4 and some pointers for possible extensions are discussed in Section 9.5.

9.2 Communication annotations for Charlotte

9.2.1 Introduction

Charlotte [27] is a system that addresses the difficulties of executing programs in a WAN. Some of the most prominent problems are heterogeneity, security, scalability, remotely installing programs, high communication latencies, and hiding the inherent unreliability of using widely dispersed resources.

The Java execution environment along with the Java programming language [92] successfully address the heterogeneity and security concerns and remove the need to install programs remotely other than standard execution environments (typically, a Java-enabled WWW browser). These capabilities make Java a prime choice for building environments to execute parallel programs distributed over the Internet or, more precisely, the World Wide Web.

A number of research efforts use Java to build such an environment. Some of them use message passing interfaces, others provide Distributed Shared Memory (DSM) semantics. Charlotte is such a DSM system—it uses a reliable parallel machine as programming model and the runtime system implements this model on top of unreliable machines, very similar to Calypso. Main characteristics of Charlotte are easy programmability, fault tolerance with regard to crash faults of workers, and adaptive parallelism that makes use of slow machines. Additionally, Charlotte is completely implemented in pure Java and runs on standard Java virtual machines. Unlike Calypso, Charlotte provides an object-based shared memory for objects of certain distributed classes.

But compared to simpler message passing systems, Charlotte suffers from a high overhead for maintaining correct memory semantics. In a message passing system, this overhead is avoided by having the programmer provide precise information which data is transferred where and when—a task that is automated by a DSM system. Additionally, it is a difficult task for a programmer to provide Charlotte-like capabilities like fault tolerance and adaptivity using only message passing primitives. A programmer finds himself thus faced with the difficult choice between easy programmability, fault tolerance, and adaptivity provided by a DSM system like Charlotte and high performance obtainable by using simpler message passing abstractions. Therefore, a solution is needed that maintains Charlotte's advantages for the programmer and improves its efficiency to be competitive with message passing systems; of course, such a solution should be implemented in pure Java as well. The need for such a solution is particularly felt in a Web-based environment using Java: On the one hand, high latencies and the unavailability of hardware support for detecting memory access make low-overhead solutions necessary; on the other hand, the complexity of Web-based network computing calls for higher, fault-tolerant programming models.

In this section, an annotation-based solution is presented that reduces Charlotte's communication overhead and bridges the gap between Charlotte's DSM semantics and message passing systems by allowing stepwise refinements of Charlotte programs. In the first step, a Charlotte program can be enhanced with annotations that describe the data requirements of parallel routines. The runtime system can use this information to improve the communication efficiency; the standard Charlotte distributed classes guarantee the correctness of the computation even if the annotations are wrong. In the second step, after the annotations' correctness has been established, efficiency can be further improved by removing the consistency checks for shared data. In the third step, primitive data types can be used as a basis for sharing, additionally foregoing the overhead

of accessing objects. This last step combines efficient, low-overhead communication and direct data access with Charlotte's advantages without the programmer having to worry about low-level communication issues. This gradual incorporation of semantic knowledge closes the gap between the programmability advantages of DSM and the communication efficiency of message passing programs. It is also conceivable to generate annotations automatically by means of a data-flow analysis within a compiler.

The remainder of this chapter is organized as follows. Related work is described in Section 9.2.2, the Charlotte system itself is explained in more detail in Section 9.2.3. The annotation-based extensions for Charlotte are introduced in Section 9.2.4 and Section 9.2.5 contains experimental results for the various extensions—in both sections matrix multiplication is used as a common example. More information on this topic can be found in [130, 131].

9.2.2 Related work

Two areas of distributed computing are of interest: Work related to using Java for parallel or distributed computing and more general research into aspects of DSM systems, and in particular annotations, which has already been described in Section 3.2.3.

A number of recent projects use Java as an implementation platform for distributed computing. A PVM interface for Java is described in [77]. ATLAS [20] extends Cilk's technologies [37], e.g., hierarchical work stealing, and integrates them into Java. Unlike Charlotte, ATLAS needs daemon processes as compute servers and also makes use of native code. JavaParty [224] transparently adds remote objects to Java by introducing a new keyword `remote` that is handled by a preprocessor. JavaParty mainly targets clusters of workstations. The programmer or compiler generate code to guide data distribution and migration. JavaParty programs are very similar to Java programs; their efficiency is comparable to RMI-based implementations. The ParaWeb project [43] is concerned with providing an infrastructure for seamless access to heterogeneous computing resources. A library can be used for explicit message passing programs or, with a modified Java Virtual Machine, threads can run remotely and are presented with the illusion of a single, shared memory. Similarly, YU [311] suggests implementing a modified Java Virtual Machine on top of TreadMarks, using a distributed garbage collector. Javelin [45], similar to Charlotte in that it allows standard WWW browsers to be used, provides brokering functionalities for computational resources and adds a layer supporting the implementation of parallel programming models in Java.

In the context of responsiveness, estimating execution times of Java programs in general is an important question. This question is non-trivial in Java, owing to some peculiarities of the language like the use of garbage collection. NILSEN [214] describes the Portable Executive for Reliable Control (PERC) system that reimplements the Java virtual machine to make it suitable for real-time requirements.

9.2.3 The Charlotte system

Charlotte is an implementation of the eager scheduling and TIES concepts from Calypso and shares Calypso's basic programming model: A Charlotte program consists of alternating sequential and parallel steps. A master application (running as a stand-alone Java application) executes the sequential steps and administers the parallel steps. In such a parallel step (delimited by `parBegin()` and `parEnd()`), a number of routines are defined and picked up by any number of workers, which are applets running in a browser. The end of a parallel step is a barrier synchronization for all the routines in this step. The current Charlotte implementation does not allow nested parallel steps. The memory is logically partitioned into private (local to a routine) and shared segments; the shared memory has concurrent read, exclusive write semantics. Charlotte, as well as Calypso, deliberately chooses this conservative semantics to make the programming model as simple as possible, arguing similarly as HILL [105].

Since the Java programming model does not allow access to low-level operating system abstractions like pages and page faults, Charlotte has to use different means to implement the abstraction of a shared memory: it is realized at the data type level. For a Java primitive type like, e.g., `int`, there is a corresponding Charlotte class `Dint` (distributed int) that provides the correct distributed semantics. The actual data access happens

via member functions `get()` and `set()`, since Java does not allow operator overloading or implicit type conversion. If, upon a read access, a data item is detected to be invalid at a worker (workers have no valid data at the beginning of a parallel step), this worker sends a request for this data item to the master and declares it valid upon reception. During a write access, the object is marked as modified; all modified objects are sent back to the master at the end of a routine. Since the Java applet security restrictions impose a star-like topology on communication, this master/worker structure fits especially well. Additionally, Charlotte's memory semantics allows the use of eager scheduling just like in Calypso. As an example, Figure 9.1 shows the skeleton of a matrix multiplication program in Charlotte. A parallel step is implemented as a class derived from `Droutine`, the method `drun` of this derived class is then the actual implementation of a routine.

```
// multiply  two SizexSize matrices: C = A*B
public class Matrix extends Droutine {
  // this method is executed by the workers:
  // compute row 'myId' of C
  public void drun (int numRoutines, int myId) {
    int sum;
    for(int col=0; col<Size; col++)  {
      sum = 0;
      for(int k=0; k<Size; k++)
        sum += A[myId][k].get() * B[k][col].get();
      C[myId][col].set(sum);
    }
  }
  ...
  // this method is executed by the master:
  public void run () {
    ...
    // a parallel step with Size routines
    // (one for each row):
    parBegin();
    addRoutine (this, Size);
    parEnd();
    ...
  }
}
```

Figure 9.1: Matrix multiplication program in Charlotte (abbreviated).

It is important to point out that Charlotte only uses standard Java mechanisms and does not, like some other projects, require a modified Java Virtual Machine or low-level libraries. In particular, the distributed classes like `Dint` are standard Java classes.

9.2.4 Annotation mechanisms

Annotating routines

Requesting data from the master upon read access can be very time-consuming, particularly in a high-latency environment. Charlotte tries to amortize this overhead by copying, for each request, not only a single object but a set of objects from the master to the worker. This set is called a "page" (not to be confused with virtual memory pages). Choosing page sizes is difficult: large pages reduce the frequency of data requests, small page sizes reduce redundant copying of objects. Since Charlotte has no way of predicting which data is going to be used, any page size is merely a guess.

If, on the other hand, the programmer gave Charlotte some hints which data is actually going to be used by a routine, Charlotte could send this "read set" along with the routine itself. The advantage of doing it is twofold: there is no latency wasted for data requests and no communication bandwidth is wasted for superfluous copying of objects (if the hints are correct). If the hints turn out to be wrong, the correctness of the program is still guaranteed since a read access to data that has not been sent in advance is still detected and served by standard Charlotte mechanisms. In this sense, these hints are correctness-insensitive. Additionally, it is possible to generate a runtime warning if hints turn out to be redundant or incomplete.

Hints are given by annotating a routine: a method `dloc_read` is defined in class `Droutine` that is called by the runtime system to obtain the read set for a given routine (see Figure 9.2 for an example).[1] Since both `drun` and `dloc_read` are methods of the same object, the association between a routine and its annotations poses no problem. Similar hints can be given for the data written by a routine.

```
public class Matrix extends Droutine {
  ...
  public Locations dloc_read (int numRoutines, int myId)
  {
    // compute the read set and store it in loc
    Locations loc = new Locations();

    // all of B:
    loc.add (B);

    // row ‘‘myId” of A:
    loc.add (A[myId]);

    return loc;
  }
  ...
}
```

Figure 9.2: Annotating a Charlotte routine with its read set (based on the matrix multiplication example of Figure 9.1).

The master keeps track of which data is currently valid at which worker. Thus, if two routines with overlapping read sets are given to a worker, only the missing data is sent with the second routine.

Relying on annotations

Assuring the local availability of data at a worker is costly: essentially, an `if` statement has to be executed for every read access to a shared object and a flag has to be set for a write access. If the programmer is sure that the annotations describe the read and write behavior of a routine correctly (e.g., after sufficient testing), or if they are generated by a compiler, there is no longer any reason for this overhead since the master takes care of sending the required data and the worker knows which data to return to the master.

Saving this overhead can be accomplished by using unchecked counterparts of Charlotte's distributed classes (`Uint` instead of `Dint`, etc.), retaining the exact same interface as the correctness-guaranteeing classes. Moving from checked to unchecked classes is a mere syntactic change of class name and constructor.

[1] In the base class `Droutine`, `dloc_read` just returns `null`. The class `Locations` handles descriptions of data sets. In the current implementation, individual objects, arrays, subarrays and matrixes of primitive types and Charlotte's distributed classes can be added directly. It is also possible to extend `Locations` to handle other, application-specific classes.

Sharing primitive types

For these unchecked classes, the `get()` and `set()` methods for such unchecked data are completely trivial—there is no longer any reason to pay the overhead for their invocation. As a matter of fact, primitive data types like `int` can be used directly—the runtime system uses the annotations to move data back and forth between master and worker as needed. Thus, the shared memory semantics of Charlotte can be implemented on top of primitive data types allowing direct access without Java's high method invocation overhead (much as it would be done in a message passing program).

Unlike the unchecked classes, using primitive data types does change the interface, e.g., the `get()` and `set()` method invocations have to be removed. In addition, objects are passed by reference, primitive types by value. The semantic difference can make the transition to primitive types awkward for single variables. But since the overhead for single variables is small in either case, the main advantages lie in the use of annotations for arrays, and arrays of objects and arrays of primitive types do have the same passing semantics. Nevertheless, this step requires careful consideration.

Note that it is of course possible to mix objects of the original Charlotte classes (with or without annotations) and shared primitive types at will. This possibility allows a programmer to use Charlotte's distributed classes for data with complicated access patterns and primitive types for more straightforward data.

Additional optimizations

Since the master keeps track of which data is currently valid at a worker, it is possible to use this information for two additional optimizations.

First, the master can use the difference between a worker's valid data set and a routine's read set as a criterion for choosing which routine to give to a worker. Choosing the routine that minimizes this difference also minimizes the amount of data the master has to send for this routine assignment.[2] In a sense, a routine is given to a worker that already has "colocated" data for this routine (hence "colocation" as a short term for this heuristic). It is interesting to note that colocation turns out to be helpful for Charlotte (see Section 9.2.5), but the same technique implemented in Calypso proves to only slightly improve performance for some programs and occasionally even degrades performance. This behavior implies that Calypso's page fault-based mechanism is faster than the necessary overhead to keep track of which pages are located at which worker.

Second, for the unchecked shared objects, data movement to the workers is solely the master's responsibility. It is therefore possible to leave all the workers' local data intact at the end of a parallel step (instead of declaring them invalid as in standard Charlotte) and to overwrite them with new values only if necessary. If a program declares shared data as unchanged at the beginning of a parallel step, the master will not remove this data from the workers' valid data set and therefore not send it again. This mechanism constitutes inter-step caching. It also allows colocation to take advantage of data send in a previous step and not to be restricted to overlapping data within one step.

Discussion

The most important fact to note for these extensions is that they allow a gradual improvement of a program: From Charlotte's pure DSM to DSM plus hints to shared objects without correctness checks to sharing primitive data types whose correctness is completely based on annotations (cp. Figure 9.3). Access to these primitives types is direct without any method invocations and therefore equivalent to what is commonly used in message passing systems.

The annotations for read and write data sets of a routine do look a little like reading and writing data from and to the network. But since only the data sets are described, the programmer does not have to worry about a stream programming interface or I/O-exceptions. Additionally, only one description is necessary as opposed to code for sending and receiving data—by comparison, a message passing program actually overspecifies

[2] Assuming the routine is correctly described by the annotations. Otherwise, this is the best guess the master can make regarding the amount of communication for a given worker/routine combination.

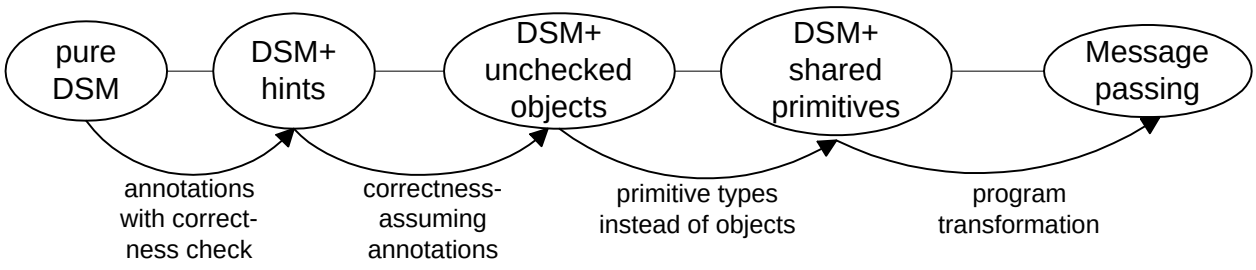

Figure 9.3: Steps between Charlotte's DSM and a message passing system.

the communication. It is possible to transform these descriptions into direct send/receive calls (corresponding to the last arrow in Figure 9.3), but the gain should be minimal; moreover, it is difficult to avoid redundant data transmission with pure message passing calls. While this approach does generate some overhead for the runtime system, the following section shows that this overhead is well invested.

All Charlotte's initial advantages like fault tolerance and adaptive parallelism are maintained —capabilities that would be laborious and error-prone to implement using message passing primitives alone. Additionally, a flexible mixture of purely DSM-based and annotation-supported objects is possible.

In comparison to related work (see in particular Section 3.2.3), C Region Library (CRL) [120] and Cid [212] are close relatives. But Charlotte's simpler programming model and the direct use of objects make these annotations easier to use for a programmer than having to worry about mapping and locking memory regions (as is done in CRL and Cid), plus having the additional possibility to use pure DSM objects. Jade's [242] annotation technique is also very similar to the approach proposed here, but it lacks the capability to mix different levels of correctness guarantees; Jade completely relies on the correctness of the given annotations.

9.2.5 Some experiments

In this section, the differences between and advantages of the various approaches are illustrated with matrix multiplication as a basis for measurements. Matrix multiplication was deliberately chosen as a problem with only a moderate ratio of computation and communication. Problems with a very high computation/communication ratio (e.g., computing prime numbers) suffer from the problems addressed by the extensions proposed here only to a much smaller degree. Problems with too small a ratio, on the other hand, are ill suited to WAN environments.

The environment used for experiments consisted of a number of PentiumPro 200 machines at the Distributed Systems Laboratory of New York University connected by a 100 Mbps Ethernet and two Pentium 90's at Humboldt University Berlin, which served as remote worker machines. A ping between these two sites typically took about 130 ms.[3] All machines were running Linux 2.0. Sun's Java Development Kit (JDK) version 1.1.3 and the Kaffe Virtual Machine version 0.92 [306] (a Java Just-In-Time (JIT) compiler) were used to run the programs. Multiplying two 200x200 matrices takes about 8.1 s on a Pentium 90 and 2.3 s on a PentiumPro 200 when using the Kaffe JIT compiler.[4]

An important question asked of an enhancement for a parallel system if the one regarding improvements in runtime. Figure 9.4 shows the runtime for a 200x200 matrix multiplication with up to four workers, measured on the local network at New York University (NYU) (all numbers are averaged over 10 runs) using the Kaffe JIT compiler. In this figure as in all the following ones, results are shown for the standard version of Charlotte, for Charlotte with checked annotations, for correctness-assuming, unchecked annotations, for the use of primitive types instead of distributed classes, and for a pure message passing implementation (implemented directly on top of Java IOStreams).

[3] As of winter 1997/98.

[4] KARL [130] also discusses results using the JDK's interpreter to run these experiments, which are considerably worse than the results of Kaffe. Additionally, since today JIT technology has matured and is commonly available, runtimes on an interpreter are of little interest and are therefore not discussed here.

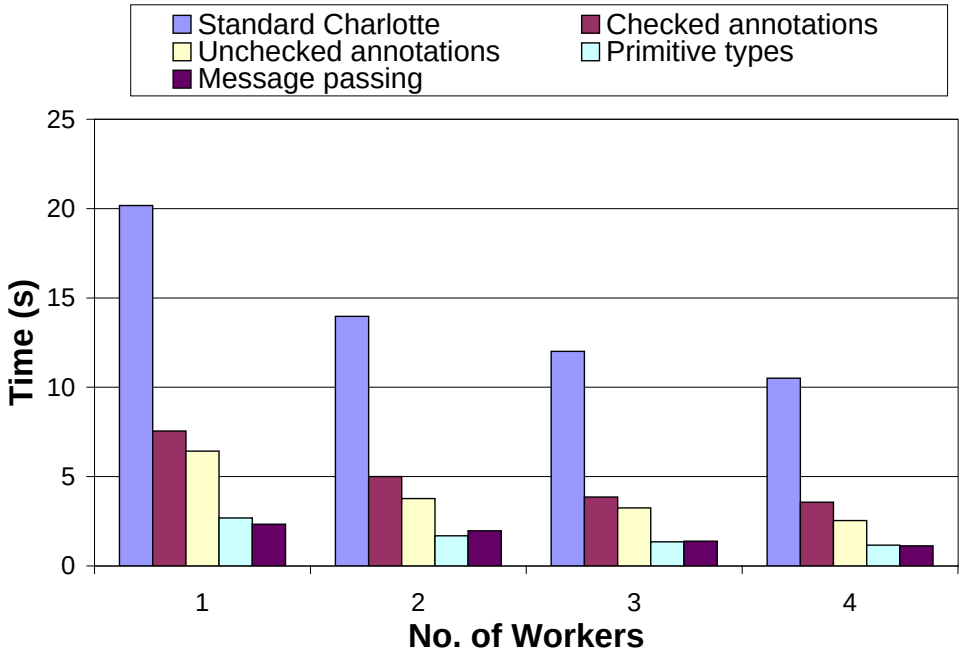

Figure 9.4: Average runtime of matrix multiplication on a local network (NYU) shown for varying number of workers and annotation levels.

The improvements in runtime using annotations are striking. Note that with shared primitive types (`int`), one worker executes almost as fast as the sequential version and shows actual speedup with two or more workers in the interpreted version; the message passing implementation suffers from only negligible overhead with one worker. Figure 9.5 shows the absolute speedup/slowdown of the compiled case compared with the sequential execution time (thus taking Charlotte's overhead into account).

It is particularly interesting to compare the runtimes needed by the different extensions of Charlotte that have been introduced in this chapter and the message passing version. The times for message passing and the Charlotte program with primitive data types are practically identical (Charlotte even outperforms message passing for two or three workers owing to better load balancing), proving the claim that with annotating Charlotte, the efficiency of message passing can be nearly met while still maintaining advantages like fault tolerance.

Figure 9.6 shows the ratios of runtimes when comparing standard Charlotte (`Dint`) with annotated Charlotte (`Dint`+A), Charlotte with unchecked distributed objects (`Uint`), and Charlotte with primitive types (`int`), respectively. The annotations make data requests unnecessary and send all the data needed for a routine in one transmission, improving runtime by about a factor of three (`Dint` vs. `Dint`+A). The `Uint` version shows another slight improvement, but the ability to forego the overhead associated with objects and to share primitive types adds another factor of two—resulting in an overall improvement of about a factor of nine over standard Charlotte (for four workers).

The runtime over connections with high latencies was tested with two workers running at Humboldt University (HU) Berlin; runtime and ratios between various methods for this setup are shown in Figure 9.7 and Figure 9.8 respectively.[5] Again it is obvious that the shared primitives version attains a performance comparable to the message passing implementation. Unfortunately, since these machines are considerably slower than the local machines, the numbers are not directly comparable and no direct conclusions concerning the respective gains for low- and high-latency environments are possible.

The optimizations in Subsection 9.2.4 were also proposed with long latencies in mind. For the example of multiplying a matrix A with two matrixes B_1 and B_2 in two consecutive parallel steps, Figure 9.9 shows the communication time using `Dint` plus annotations, additionally caching A between the two steps, and

[5] `Uint` is not shown since, as seen above, the major improvements steam from annotations and primitive data types.

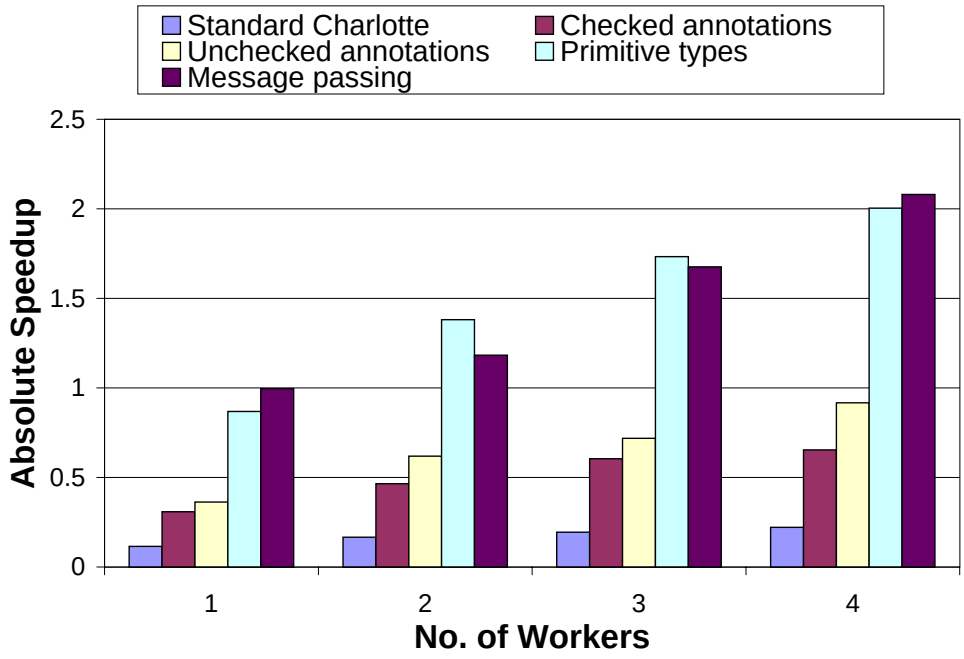

Figure 9.5: Absolute speedup/slowdown of matrix multiplication on a local network (NYU) shown for varying number of workers and annotation levels.

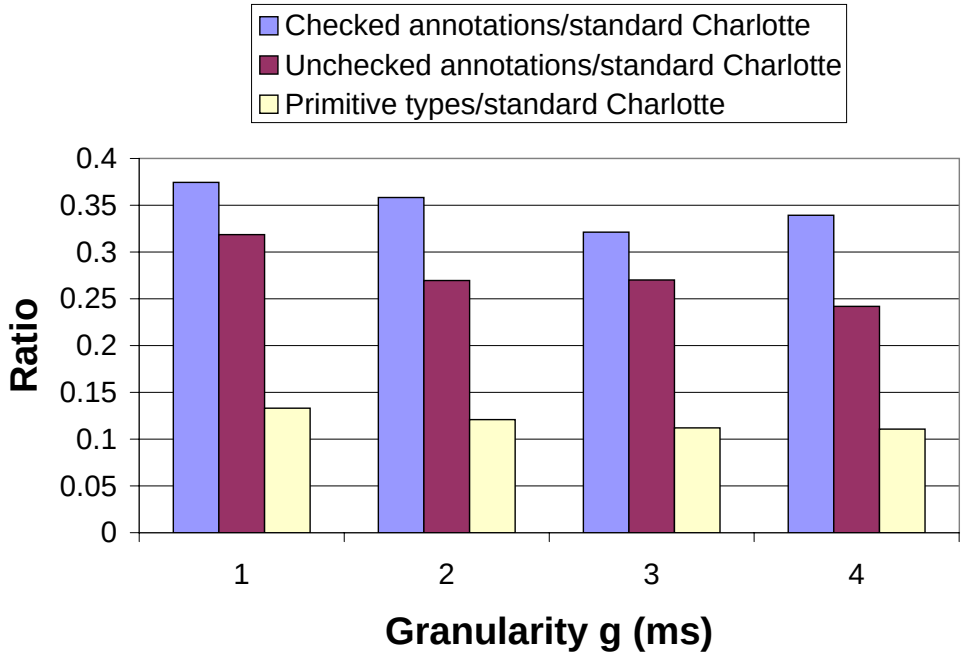

Figure 9.6: Ratio of matrix multiplication runtimes on a local network (NYU), comparing effects of various annotations levels with standard Charlotte, shown for varying number of workers.

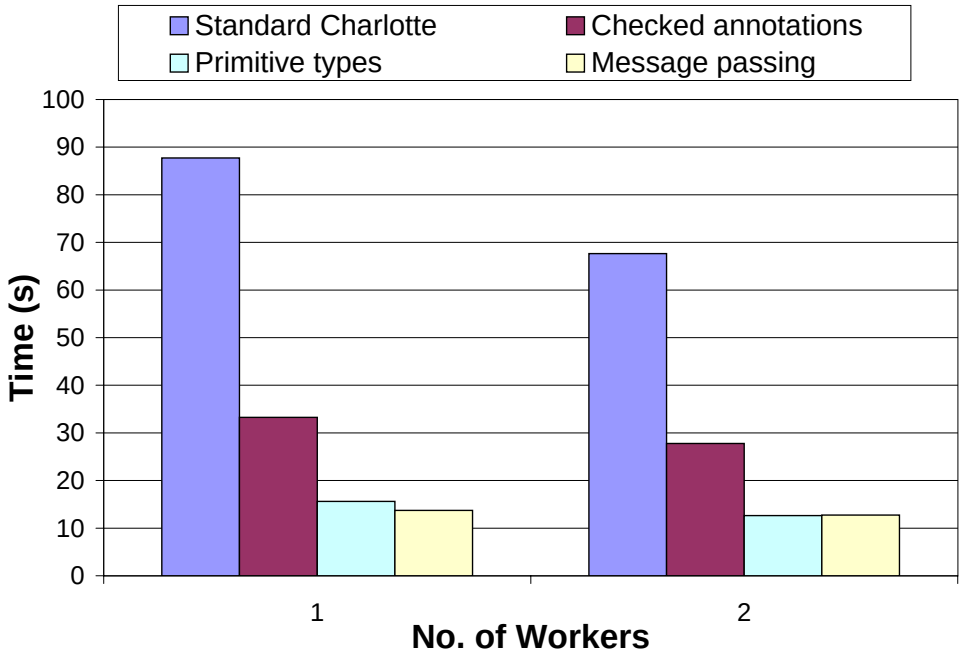

Figure 9.7: Average runtime of matrix multiplication with master at NYU and workers at HU shown for varying number of workers and annotation levels.

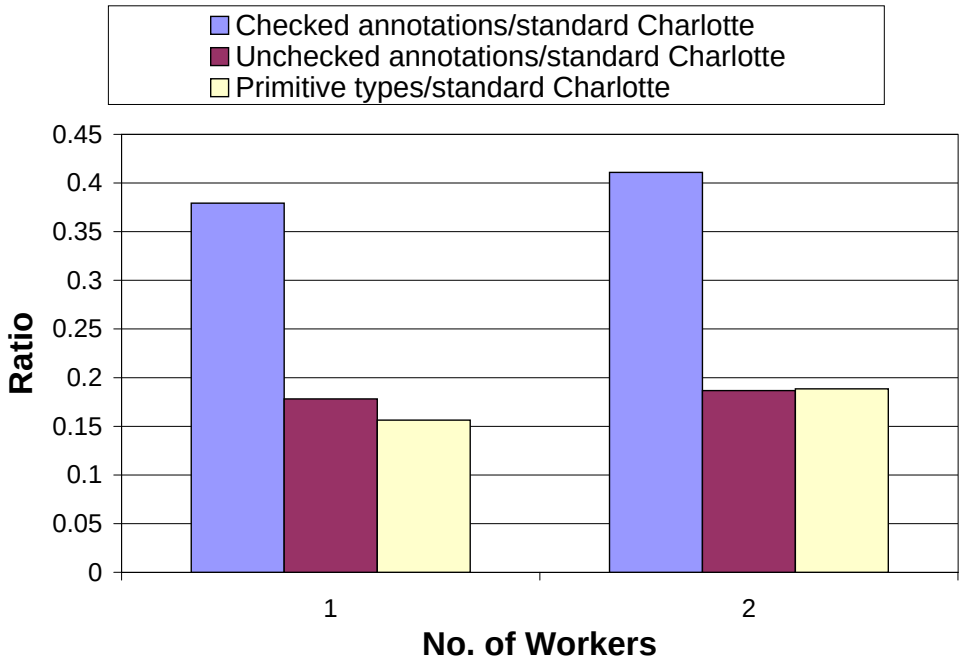

Figure 9.8: Ratio of matrix multiplication runtimes with master at NYU and workers at HU, comparing effects of various annotations levels with standard Charlotte, shown for varying number of workers.

both caching A and taking the distribution of A among the workers into account for the second parallel step (colocation). While in a LAN environment the impact of colocation is only small, for high-latency connections colocation can save up to 25% of communication time. Perhaps even more important for responsive computing is the fact that the standard deviation of colocation is roughly a factor of three smaller than the ones of the other methods.'

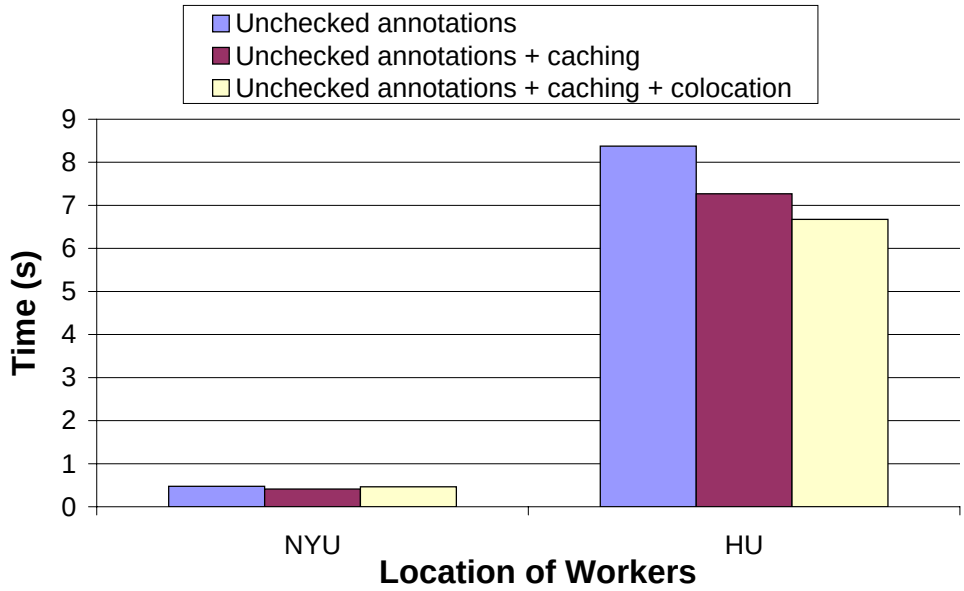

Figure 9.9: Average communication times for matrix multiplication shown for workers at NYU or HU; `Dint` plus annotations, `Dint` plus annotation and caching, and `Dint` plus annotation and caching and colocation (averaged over 1000 runs).

9.3 An infrastructure for resource allocation in the WWW

Charlotte makes it easy for a volunteer to contribute his idle CPU time to a parallel application, but it does not answer the question how a volunteer can find such an application. This problem is solved by KnittingFactory, described in much more detail in [25, 26].

One main component of KnittingFactory is a directory service. The requirements for such a directory service are slightly different from a typical name server. One requirement is to allow lookups not only from programs, but also from within a standard web browser. Another requirement is to accommodate highly dynamic registration and deregistration of processes and to consider the topological structure of processes to favor applications that are close to volunteers.

The requirement to use browsers as lookup tools implies that such a directory server should be integrated into the Web infrastructure. In KnittingFactory, applications looking for workers can register with a directory service by sending standard Hypertext Transfer Procotol (HTTP) requests to well known KnittingFactory servers. These servers store the requests along with information about peer KnittingFactory servers.

A volunteer looking for work directs his browser to such a KnittingFactory server and retrieves an Hypertext Markup Language (HTML) page from this server that includes a list of known applications and peer KnittingFactory servers and also contains a small Javascript [79] program. This program inspects the page to check if there are any applications looking for work known to this server. If so, the browser is redirected to the Uniform Resource Locator (URL) of the application (corresponding to, e.g., a Charlotte program) and, transparently for the user, downloads the corresponding applet and starts to execute it. If no application is found, the Javascript program constructs a new URL from the current URL, starting with a peer server, appends the name of the current server to it, and redirects the browser to this new URL. This passes on state information between several pages and allows the Javascript program to implement different search strategies.

One strategy would be breadth-first search, favoring topologically near servers (as defined by the topology of the server graph). Such topological considerations are difficult when using other directory services. Additionally, this client-based search implementation moves the actual search from the servers to the clients and can therefore be regarded as an implementation technique of the Smart Client concept advocated by YOSHIKAWA et al. [309].

KnittingFactory also removes another limitation of typical Java applications. For security reasons, a Java applet is by default only allowed to open network connections to the machine from which it was downloaded. Since downloading requires an HTTP server, a machine running such a server can easily become a bottleneck if multiple (e.g., Charlotte) applications are running on it. On the other hand, it is often impractical to install a complete HTTP server on all available machines. To overcome this limitation, KnittingFactory provides a core HTTP server functionality that can be easily integrated into any Java application. Using this integrated server, a Java application can be started on any machine, optionally registers itself with one or more KnittingFactory servers, and awaits requests for applets. This easy access to core HTTP functionality increases the flexibility of Java applications.

9.4 Conclusions

Problems arising from applying Calypso's techniques to wide area network environments, in particular in the context of reducing communication overhead, have been considered in this chapter. In Section 9.2, possibilities for improving Charlotte's efficiency by means of annotating parallel routines with their communication dependencies have been proposed. These annotations reduce the communication overhead of a Charlotte program, enable the use of simpler memory management techniques, and can additionally be interpreted as bridging the gap between the DSM semantics of Charlotte and simpler, yet more efficient message passing systems.

These annotations can have the character of hints, allowing the runtime system to improve communication efficiency while still guaranteeing the correctness of a program. They can also be used as a precise description of read and write sets, which allows the sharing of primitive types like `int` across multiple machines. The stepwise nature of this concept enables a programmer to gradually incorporate knowledge about a program's behavior into the code and to freely mix pure DSM objects with annotation-based objects or shared primitive types.

Sharing primitive types results in data access efficiency usually found only in message passing systems or hardware-supported DSM systems. By building on top of Charlotte, this efficiency is now available for Java-based Web computing without putting the burden of low-level communication primitives on a programmer while properties that are crucial for Web computing, e.g., fault tolerance, are maintained. In this sense, advantages from both DSM systems and message passing are incorporated in this concept.

The practicability and ease of use of this approach has been shown with matrix multiplication as an example. A number of measurements substantiate the claim to vastly improved performance—runtime improvements of up to a factor of nine over standard Charlotte and competitive with a pure message passing implementation were observed. These results show that with modest overhead for programmer and runtime system, even problems of only moderate granularity can be efficiently solved in a Java-based DSM programming environment.

To support Charlotte's need to find volunteers contributing to the computation, and in general to make Web-based applications more feasible, KnittingFactory provides an easy-to-use, Web-based directory service and a mechanism to execute Java applications on any host without the need for an external HTTP server.

9.5 Possible extensions

There are a number of possible extensions to this work. With regard to responsiveness, these annotations provide hints to the runtime system about the communication overhead of the parallel execution. Adding further information about the execution times of routines along with a fault model is simple, allowing the

runtime system to make on-line estimations of the responsiveness, based on the techniques developed in Chapter 5. Combining it with emerging technologies for real-time Java adds to the predictability of the program execution. Additionally, the knowledge about communication requirements can be used in concert with information about the network status (as obtained from systems like the Network Weather Service [307] or the Network Status Predictor [146]) to further improve the responsiveness of a program. If the runtime system decides that it is unlikely to meet a requested deadline, it can request additional resources from a system like KnittingFactory (see Section 9.3). Conversely, resources can be released if the probability of meeting the deadline is sufficiently high.

Another extension is studying the impact of problem size and communication/computation ratio on the relative performance of the various annotation levels. Generating the annotations by a compiler-based data-flow analysis would also be most interesting. Overlapping computation with communication is an orthogonal issue: Coordinated execution of multiple workers within one browser is an obvious approach to this problem.

Taking a long-term perspective, it must be noted that JIT compilers for Java still do not deliver the performance that has been expected from them when they first become popular. This shortcoming makes Java somewhat less attractive for implementing metacomputing systems. Nevertheless, Java is still attractive as a coordination language for such metasystems. In such a scenario, Java programs concert the execution of lower-level programs at different sites, probably larger facilities based on supercomputers. For such installations, the deployment of, e.g., ATM-based virtual circuits becomes viable, providing guaranteed communication Quality of Service. Such a scenario would allow to reconsider responsive computing in wide area networks from a new perspective.

"Begin at the beginning", the King said gravely, "and go on till you come to the end: then stop."
– Lewis Caroll

Chapter 10

Conclusions and Future Work

10.1 Conclusions

Clusters of standard, off-the-shelf workstations have become more and more popular for parallel computing and are currently a viable alternative for custom-built high-performance systems (e.g., parallel supercomputers) in many application areas. This dissertation has concentrated on additional challenges of parallel computing beyond mere performance: timely and dependable execution of parallel programs. Given the prevalent focus on high performance in cluster computing research, the questions of dependability and timeliness, and in particular their combination, have received comparatively scant attention. To address these issues, questions of scheduling analysis, fault tolerance, resource management and communication should be answered and expressed in a concise metric; solutions should be compatible with the commodity nature of cluster-based systems.

Such a concise metric has been found in the existing notion of responsiveness, which has been refined to fit the needs of this work. Responsiveness is the probability of correctly completing a service before or at a given deadline, even in the presence of faults. If a deadline is not given, then the distribution of the service's response time is an appropriate metric. Based on the large amount of work in cluster computing, the systems of the Milan project, namely Calypso and Charlotte, have been selected to serve as a case study from which concrete responsiveness needs of parallel computing could be extracted. Four such needs have been identified: a response time analysis, dealing with single points of failure, providing guaranteed access to resources, namely CPU time, and reducing communication overhead.

The response time distribution of the eager scheduling mechanism employed by the Milan systems has been analyzed under some general assumptions about the behavior of machines and programs. The general analysis considers arbitrary probabilistic distributions of routine execution times and machine lifetimes and derives the response time distribution from these assumptions. The general solution is only of limited practical value since its numerical complexity is large, and simulations are often preferable over analytical approaches—hence carefully restricted assumptions about the program behavior (which implies guidelines for program design and implementation) are necessary. If the assumptions are restricted to fixed routine execution times, the analytical solution is competitive with simulation and practically feasible. Moreover, under both sets of assumptions, analytical and simulation results show an correspondence.

To remove a single point of failure, two popular mechanisms for fault tolerance, checkpointing and replication, have been investigated with regard to responsiveness. For checkpointing, a simple yet general theoretical solution of the problem of maximizing responsiveness by an appropriate choice of the checkpointing interval has been given. This analytical solution has been validated by experiments with a Calypso version extended by checkpointing functionality—the analytically predicted optimal checkpointing interval matches the one found in experiments (as close as stochastic claims can be made). These experiments have shown that checkpointing is a viable means to ensure that parallel programs meet their deadlines with high probability. Checkpointing

has the additional valuable property that the responsiveness is actually fairly robust with respect to the employed checkpointing interval, as long as it is in the vicinity of the optimal interval (also as could be expected from the analysis).

Based on replication, a general-purpose system, Fault-Tolerant Distributed I/O (FT-DIO), has been introduced that increases application fault tolerance of existing legacy software by observing their input/output behavior. FT-DIO is also characterized by a flexible configuration and adaption of the fault-tolerance level to the needs of the application, even at runtime. Experiments with FT-DIO have shown that not only the fault model but also the replication mechanism have a large impact on performance, in particular, that removing a single point of failure incurs high overhead. To assess the suitability of FT-DIO for responsiveness, the Totem protocol, which is a key component of FT-DIO, has been investigated experimentally; the theoretical results for Totem's predictability have been confirmed for simple fault models. However, for more complicated fault models or in the presence of additional background load on some machines, Totem's predictability suffers considerably. On the basis of FT-DIO, a replicated version of Calypso has been designed. Experiments have shown that replication increases the responsiveness of a Calypso program under heavy fault injection; compared with checkpointing, however, it does not perform favorably in the concrete experiments that have been considered. These experiments indicate that for practical environments a combination of checkpointing with a modest degree of replication (i.e., a duplex system) promises a high degree of responsiveness.

The need of a parallel program for guaranteed amounts of resources to complete execution in time has been addressed by a resource management scheme that both conforms to the standards used in clusters and is appropriate for the use with parallel programs. Compared with existing resource management systems, the one presented here combines guaranteed CPU share with temporally coordinated execution of distributed processes (coscheduling) without modifying the underlying operating system or hardware. Experiments have shown that different parallel programming models have different synchronization requirements; in particular that for BSP-style programs coscheduling is both necessary (runtime improvements of over one order of magnitude are observable) and feasible, but also that coscheduling can be harmful to the performance of master/worker style programs (as represented, e.g., by Calypso).

In the last chapter, the impact of reducing communication between distributed parts of a program has been investigated, which is especially important for parallel computing in wide area environments. An annotation-based solution has been presented that reduces the communication overhead of Charlotte programs, enables the use of simplified memory management mechanisms, and can serve as a first stepping stone towards responsive parallel computing in these complex settings. Additionally, these annotations can be interpreted as bridging the semantic gap between distributed shared memory and message passing systems. Experiments show that these annotations improve the efficiency of Charlotte programs by up to a factor of nine.

Based on the case of a concrete system and its requirements, some general models of program behavior with respect to responsiveness have been derived in this dissertation; analytical solutions for scheduling and checkpointing and standard-conforming solutions to questions regarding replication and resource management have been proposed and corroborated by experiments—over 10,000 machine hours were used to run numerical analysis and experiments. The need to carefully select assumptions has become apparent, resulting in guidelines for the development of programs suitable for responsive execution. Also, handling problems at different abstraction levels is of paramount importance: No solution for timeliness or dependability at any single abstraction level is sufficient, since statements made at one level can be jeopardized by system properties at another level. This dissertation has made the first step towards an integrated treatment of multiple levels, facilitated by applying responsiveness as a single metric, but research towards integration must still continue. Much of the techniques proposed here are also applicable in other environments and represent both theoretical and practical contributions to questions in responsive execution of parallel programs. However, much work remains to be done and some directions for future research are considered in the following section.

10.2 Future work

10.2.1 Parallel computing

The use of clusters of workstations for parallel computing has already matured considerably over the years. Much research has made clusters of workstations almost ready for everyday deployment for high-performance applications, and many remaining questions regard usability and practicality issues. Fundamental issues that are perhaps not yet sufficiently understood are the concept of locality of reference in both space and time and its consequences for programming models and system architecture (in particular, memory hierarchies) and the realization of powerful parallel I/O capabilities in a cluster of standard workstations. Also with respect to responsiveness many questions are still open.

One pressing question here is the integration of the many individual solutions that exist in Quality of Service for networks, endsystem Quality-of-Service architectures, user-level network access, scheduling analysis and programming models into a coherent solution that truly supports end-to-end responsiveness for high-performance computing. Integration is necessary since all these mechanisms in isolation are insufficient: Quality-of-Service networks are worthless if not supported by an Quality-of-Service-aware operating system, both are worthless if network access is too slow or if it is too difficult to specify requirements, which in turn are useless if faults are not accounted for. Quality of Service must also be provided to application scheduling, in particular if the application uses user-level access to the network interface without operating system intervention. While all the individual mechanisms are perhaps sufficiently understood, there is a pressing need to integrate all of them.

Such end-to-end support for Quality of Service, in particular responsiveness, ties in with a number of different other research issues. One is the question of appropriate programming models that allow to express service properties and requirements, ideally combined with a flexible choice of alternatives, in an easy and succinct manner. Tunable programs (programs that can tradeoff resources over the course of their execution) as have been described in the Calypso context [49] are a step in this direction. Similarly, resources must be managed in a dynamic and flexible way. To do so, it is necessary to quickly judge the consequences of (re-) allocating resources to a parallel program. Speed can here be more important than precision. Fast approximation algorithms are necessary to allow a resource management system to make informed decisions. Fast decisions are also important if user feedback has to be integrated—an example would be applications that interactively compute complex simulations or numerically solve systems of equations and visualize the results in real time. Also, it would be beneficial to a programmer if the operating or runtime system could automatically detect Quality-of-Service requirements of an application without an explicit need to specify them; this ties again in with tunable programs.

Evidently, there are still a number of open questions in parallel cluster computing. But clusters of workstations are increasingly used with other objectives than parallel high-performance computing as well.

10.2.2 Availability in open system environments

One such objective that currently experiences very active interest is the provisioning of high availability to standard software. This problems typically appears in the context of servers running mission-critical applications like middle tiers of three-tier applications that must be operational at all time. Clusters of workstations are an attractive platform for such applications owing to the simple possibility of adding resources, which also entails redundancy, practically without limits. The main point here is often not only to provide additional fault tolerance against hardware failures (which is sometimes at an already acceptable level) but to provide additional functionality like scalability or ability of on-line upgrades of software, as well as protection against software failures.

A serious challenge here is the desirable transparency of any solution. As the discussion in Section 3.3.3 and Section 7.2.2 has shown, there are a number of systems that provide some of this functionality. But most of them are still laboring under a number of limitations, for example the restriction to state-free applications. This does cover many practically relevant applications like servers for the World Wide Web, however, it still leaves room for many extensions. One extension is not only to provide high availability for a Web server, but

truly predictable behavior for an end user. FT-DIO, introduced in Section 7.2, also addresses this problem, but is limited to a certain class of applications. Again, this raises many questions having to do with integration of existing individual solutions (e.g., network Quality of Service on the one hand and highly available application services on the other hand). It might not be possible to provide firm guarantees about the service behavior to an end user (because of the combinatorial explosion inherent in large network configurations), but it raises questions regarding, e.g., tradeoffs between the predictability of a service and its costs.

More generally, high responsiveness is desirable not only for specific environments like the Web, but for open systems in general. A typical example for an infrastructure for such open systems is CORBA; ideally, CORBA should ensure fault tolerance for applications running on top of it. This can be considered to be a truly new challenge since only very little knowledge about the state of such open environments is available. It is also linked with the problem of determining the behavior of a composed system, since basically all practically relevant systems are of a size that there is no other possibility but to compose them out of simpler parts. Even more preferable are components that maintain given properties when composed with other systems (where some restrictions on the compositional process might prove necessary). To this end, proper description mechanisms of the behavior of components with respect to composability are required.

The overall problem is therefore a means to enable open systems, consisting of many individual components, to deliver services in a predictable, timely and dependable manner. Given the difficulties of reengineering large software systems, it is necessary to develop solutions for legacy software. Problems of service interaction (where two services behave differently when combined or isolated, or even do not have a unique semantics at all [142]) complicate this problem further.

References

[1] T. Abdelzaher, A. Shaikh, F. Jahanian, and K. Shin. RTCast: Lightweight Multicast for Real-Time Process Groups. In *Proc. 2nd Real-Time Technology and Applications Symp.*, pages 250–259, Boston, MA, June 1996. `ftp://rtcl.eecs.umich.edu/outgoing/zaher/rtas96.ps`. Also as TR CSE-TR-291-96, Department of Computer Science and Engineering, University of Michigan.

[2] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12): 66–76, 1996.

[3] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proc. 22nd Intl. Symp. on Computer Architecture*, pages 2–13, Santa Margherita Ligure, Italy, June 1995. `ftp://ftp.cag.lcs.mit.edu/pub/papers/isca95.ps.Z`.

[4] A. Alles. ATM Internetworking. Cisco Systems, Inc., `http://www-europe.cisco.com/warp/public/614/12.html`. Abridged version at Proc. Engineering InterOp, Las Vegas, NV, 1995.

[5] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. The Totem Single-Ring Ordering and Membership Protocol. *ACM Trans. on Computer Systems*, 13(4):311–342, 1995.

[6] Y. Amir and J. Stanton. The Spread Wide Area Group Communication System. Technical Report CNDS 98-4, Department of Computer Science, John Hopkins University, 1998. `ftp://ftp.cnds.jhu.edu/pub/papers/spread.ps.gz`.

[7] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, 1996.

[8] T. Anderson and P. A. Lee. *Fault Tolerance, Principles and Practice*. Prentice Hall, Inc., Englewood Cliffs, NJ, 1982.

[9] T. E. Anderson, D. E. Culler, and D. A. Patterson. A Case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, 1995.

[10] J. N. C. Arabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, and P. Stephan. Dome: Parallel Programming in a Heterogeneous Multi-User Environment. Technical Report CS-95-137, Department of Computer Science, Carnegie Mellon University, April 1995.

[11] S. Araki, A. Bilas, C. Dubnicki, J. Edler, K. Konishi, and J. Philbin. User-Space Communication: A Quantitative Study. In *Proc. Supercomputing*, Orlando, FL, November 1998. `http://www.cs.princeton.edu/shrimp/Papers/sc98USC.ps` and `http://www.supercomp.org/sc98/TechPapers/sc98_FullAbstracts/Bilas820/index.htm`.

[12] C. M. Aras, J. P. Kurose, D. S. Reeves, and H. Schulzrinne. Real-time Communication in Packet-Switched Networks. *Proc. of the IEEE*, 82(1):129–139, 1994. `ftp://gaia.cs.umass.edu/pub/Aras9401:Real.ps.gz`.

[13] A. C. Arpaci-Dusseau and D. E. Culler. Extending Proportional-Share Scheduling to a Network of Workstations. In *Proc. Intl. Conf. on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, NV, June 1997. `http://now.cs.berkeley.edu/Papers2/Postscript/pdpta.ps`.

[14] A. C. Arpaci-Dusseau, D. E. Culler, and A. M. Mainwaring. Scheduling with Implicit Information in Distributed Systems. In *Proc. SIGMETRICS '98/PERFORMANCE '98 Joint Conf. on Measurement and Modeling of Computer Systems*, pages 233–243, Madison, WI, June 1998.

[15] Y. Aumann, Z. Kedem, K. Palem, and M. Rabin. Highly Efficient Asynchronous Execution of Large-grained Parallel Programs. In *Proc. 34th Ann. Symp. on the Foundations of Computer Science*, pages 271–280, Palo Alto, CA, November 1993.

[16] C. Aurrecoechea, A. T. Campbell, and L. Hauw. A Survey of QoS Architectures. *Multimedia Systems Journal, Special Issue on QoS Architecture*, 6(3):138–151, 1998. `http://www.ctr.columbia.edu/~campbell/`

andrew/publications/publications.html.

[17] W. E. Baker, R. W. Horst, D. P. Sonnier, and W. J. Watson. A Flexible ServerNet-based Fault-tolerant Architecture. In *Proc. 25th Symp. on Fault-Tolerant Computing*, pages 2–11, Pasadena, CA, June 1995.

[18] H. E. Bal and M. F. Kaashoek. Object-Distribution in Orca using Compile-Time and Run-Time Techniques. In *Proc. Conf. on Object-Oriented Programming Systems, Languages and Applications*, pages 162–177, Washington, D.C., September 1993.

[19] H.E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Ruhl, and M.F. Kaashoek. Performance Evaluation of the Orca Shared Object System. *ACM Trans. on Computer Systems*, 16(1):1–40, 1998. ftp://ftp.cs.vu.nl/pub/amoeba/orca_papers/tocs98.ps.Z.

[20] J. E. Baldeschwieler, R. D. Blumofe, and E. A. Brewer. ATLAS: An Infrastructure for Global Computing. In *Proc. 7th SIGOPS European Workshop: Systems Support for Worldwide Applications*, Connemara, Ireland, September 1996. http://www.cs.utexas.edu/users/rdb/papers/SIGOPS96.ps.gz or http://gunpowder.stanford.edu/sigops96/papers/baldeschwieler.ps.

[21] S. Balji, R. Menon, F. Ansari, J. Keinig, and A. Sheth. UTIME — Micro-Second Resolution Timers for Linux. Project homepage at http://hegel.ittc.ukans.edu/projects/utime/index.html, April 1998.

[22] A. Baratloo. *Metacomputing on Commodity Computers*. PhD thesis, New York University, New York, NY, May 1999.

[23] A. Baratloo, P. Dasgupta, and Z. M. Kedem. CALYPSO: A Novel Software System for Fault-Tolerant Parallel Processing on Distributed Platforms. In *Proc. 4th Intl. Symp. on High-Performance Distributed Computing*, pages 122–129, Washington, D.C., August 1995.

[24] A. Baratloo, A. Itzkovitch, Z. Kedem, and Y. Zhao. Mechanisms for Just-in-Time Allocation of Resources to Adaptive Parallel Programs. In *Proc. 13th Intl. Parallel Processing Symp./10th Symp. on Parallel and Distributed Processing*, pages 506–512, San Juan, Puerto Rico, April 1999.

[25] A. Baratloo, M. Karaul, H. Karl, and Z. Kedem. KnittingFactory: An Infrastructure for Distributed Web Applications. Technical Report TR 1997-748, Department of Computer Science, New York University, New York, NY, November 1997.

[26] A. Baratloo, M. Karaul, H. Karl, and Z. Kedem. An Infrastructure for Network Computing with Java Applets. *Concurrency: Practice and Experience*, 10(11–13):1029–1041, 1998.

[27] A. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the Web. In *Proc. 9th Intl. Conf. on Parallel and Distributed Computing Systems*, pages 181–188, Dijon, France, September 1996.

[28] M. Barborak, M. Malek, and A. Dahbura. The Consensus Problem in Fault-Tolerant Computing. *ACM Computing Surveys*, 25(2):171–220, 1993.

[29] B. Barrow. A Lesson in Megabytes. *IEEE Standards Bearer*, page 5, January 1997. http://physics.nist.gov/cuu/Units/binary.html.

[30] A. Basu, M. Welsh, and T. von Eicken. Incorporating Memory Management into User-Level Network Interfaces. In *Proc. Hot Interconnects V*, Palo Alto, CA, August 1997. http://www2.cs.cornell.edu/U-Net/papers/unetmm.pdf Also as TR 97-1620, Department of Computer Science, Cornell University.

[31] B. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In *Proc. COMPCON 93*, pages 528–537, San Francisco, CA, February 1993. http://www.cs.cmu.edu/afs/cs/project/midway/WWW/CompCon93.ps. Also as TR CMU-CS-93-119, Department of Computer Science, Carnegie Mellon University.

[32] R. Bhoedjang, T. Rühl, and H. Bal. User-Level Network Interface Protocols. *IEEE Computer*, 31(11):53–60, 1998.

[33] B. Bieker, G. Deconinck, E. Maehle, and J. Vounckx. Reconfiguration and Checkpointing in Massively Parallel Systems. In *Proc. 1st European Dependable Computing Conf.*, pages 353–370, Berlin, October 1994.

[34] K. P. Birman. The Process Group Approach to Reliable Distributed Computing. *Comm. of the ACM*, 36(12): 36–53, 1993.

[35] R. Bjoernson. *Linda on Distributed Memory Multiprocessors*. PhD thesis, Yale University, New Haven, CT, 1992.

[36] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services. RFC 2475, December 1998.

[37] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. *J. of Parallel and Distributed Computing*, 37(1):55–69, 1996. http://www.cs.utexas.edu/users/rdb/papers/JPDC96.ps.gz.

[38] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proc. 21st Ann. Intl. Symp. on Computer Architecture*, pages 142–153, Chicago, IL, April 1994. `http://www.cs.princeton.edu/shrimp/Papers/isca94VMMC.ps`.

[39] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet — A Gigabit-per-Second Local-Area-Network. *IEEE Micro*, 15(1):29–36, 1995.

[40] W. G. Bouricius, W. C. Carter, D. C. Jessep, P. R. Schneider, and A. B. Wadia. Reliability Modeling for Fault-Tolerant Computers. In *Proc. 1st Intl. Symp. on Fault-Tolerant Computing*, pages 60–63, June 1971. Reprinted in [259].

[41] R. Braden, D. Clark, and S. Shenker. Integrated Services in the Internet Architecture: an Overview. RFC 1633, June 1994.

[42] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReSerVation Protocol (RSVP) — Version 1 Functional Specification. RFC 2205, September 1997.

[43] T. Brecht, H. Sandhu, M. Shan, and J. Talbot. ParaWeb: Towards World-Wide Supercomputing. In *Proc. 7th SIGOPS European Workshop: Systems Support for Worldwide Applications*, pages 181–188, Connemara, Ireland, September 1996. `http://cs.yorku.ca/~brecht/papers/html/paraweb/paraweb.html`.

[44] T. C. Bressoud. TFT: A Software System for Application-Transparent Fault Tolerance. In *Proc. 28th Intl. Symp. on Fault-Tolerant Computing*, pages 128–137, Munich, Germany, June 1998.

[45] P. Cappello, B. Christiansen, M. F. Ionescu, M. O. Neary, K. E. Schauser, and D. Wu. Javelin: Internet-Based Parallel Computing Using Java. *Concurrency: Practice and Experience*, 9(11):1139–1160, 1997.

[46] N. Carriero. *Implementing Tuple Space Machines*. PhD thesis, Yale University, New Haven, CT, 1987.

[47] J. B. Carter, J. K. Bennet, and W Zwaenepoel. Implementation and Performance of Munin. In *Proc. 13th Symp. on Operating System Principles*, pages 152–164, Pacific Grove, CA, October 1991. `http://www.cs.rice.edu/~willy/papers/sosp91.ps.gz`.

[48] K. M. Chandy, J. C. Browne, C. W. Dissly, and W. R. Uhrig. Analytic Models for Rollback and Recovery Strategies in Data Base Systems. *IEEE Trans. on Software Engineering*, 1(1):100–110, 1975.

[49] F. Chang, V. Karamcheti, and Z. Kedem. Exploiting Application Tunability for Efficient, Predictable Parallel Resource Management. In *Proc. 13th Intl. Parallel Processing Symp./10th Symp. on Parallel and Distributed Processing*, pages 749–758, San Juan, Puerto Rico, April 1999.

[50] R. Chapman. *Static Timing Analysis and Program Proof*. PhD thesis, University of York, York, Great Britain, 1995. `ftp://ftp.cs.york.ac.uk/reports/YCST-95-05.tar.Z`.

[51] A. Chien, M. Hill, and S. Mukherjee. Design Challenges for High-Performance Network Interfaces. *IEEE Computer*, 31(11):42–45, 1998.

[52] A. Chien, M. Lauria, R. Pennington, M. Showerman, G. Iannello, M. Buchanan, K. Hane, L. Giannini, G. Koenig, S. Krishnamurthy, Q. Liu, S. Pakin, and G. Sampemane. Design and Evaluation of an HPVM-based Windows NT Supercomputer. Accepted for publication in Special Combined Issue of Parallel Processing Letters and Intl. Journal of High-Performance and Scientific Applications, `http://www-csag.ucsd.edu/papers/csag/external/bbfarm.ps`, 1999.

[53] H.-H. Chu and K. Nahrstedt. A Soft Real Time Scheduling Server in UNIX Operating System. In *Proc. Interactive Distributed Multimedia Systems and Telecommunication Services*, pages 153–162, Darmstadt, Germany, September 1997.

[54] P. W. Ciarfella. The Totem Protocol Testbed. Master's thesis, Department of Computer Science, University of California, Santa Barbara, CA, 1993.

[55] E. Çinlar. *Introduction to Stochastic Processes*. Prentice Hall, Inc., Englewood Cliffs, NJ, 1975.

[56] D. D. Clark, V. Jacobsen, J. Romkey, and H. Salwen. An Analysis of TCP Processing Overhead. *IEEE Communications Magazine*, 27(6):23–29, 1989.

[57] D. Cohen and C. Seitz. M2M-OCT-SW8: Octal 8-Port Myrinet-SAN Switch Recommended Topologies. White Paper, `http://www.myri.com/myrinet/topology/OCT-SW8.pdf`, June 1997.

[58] D. Collins and C. Beauregard. Microsoft and Compaq Annouce Windows 2000 Initiative for System Area Networking. Press release. `http://www.tandem.com/PRES_REL/WINSOKPL/WINSOKPL.HTM`, December 1998.

[59] K. H. Connelly and A. A. Chien. FM-QoS: Real-time Communication Using Self-Synchronizing Schedules. In *Proc. Supercomputing*, San Jose, CA, November 1997. `http://www.supercomp.org/sc97/proceedings/TECH/CONNELLY/CONNELLY.PS`.

[60] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems Concepts and Design*. Addison Wesley, Cambridge, MA, 1994.

[61] F. Cristian. Probabilistic Clock Synchronization. *Distributed Computing*, 3(3):146–158, 1989.

[62] F. Cristian. Understanding Fault-Tolerant Distributed Systems. *Comm. of the ACM*, 34(2):57–78, 1991.

[63] D. E. Culler, R. Liu, L. T. Martin, and C. Yoshikawa. Assessing Fast Network Interfaces. *IEEE Micro*, 16(1): 35–43, 1996. `http://now.cs.berkeley.edu/Papers/Papers/Logp_Micro.ps`.

[64] P. Dasgupta, Z. M. Kedem, and M. O. Rabin. Parallel Processing on Networks of Workstations: A Fault-Tolerant, High Performance Approach. In *Proc. 15th Intl. Conf. on Distributed Computing Systems*, pages 467–474, Vancouver, CA, May 1995.

[65] Z. Deng, J. W.-S. Liu, and J. Sun. A Scheme for Scheduling Hard Real-Time Applications in Open System Environment. In *Proc. 9th Euromicro Workshop on Real-Time Systems*, pages 191–199, Toledo, Spain, June 1997. `http://pertsserver.cs.uiuc.edu/papers/DeLi97b.ps`.

[66] Z. Deng, Jane W.-S. Liu, L. Zhang, M. Seri, and A. Frei. An Open Environment for Real-Time Applications. Technical Report UIUCDCS-R-98-2052, University of Illinois, Urbana-Champaign, IL, June 1998. `http://pertsserver.cs.uiuc.edu/papers/DeL98.ps`.

[67] D. Dolev and D. Malki. The Transis Approach to High-Availability Cluster Communication. *Comm. of the ACM*, 39(4):64–70, 1996.

[68] S. R. Donaldson, J. M. D. Hill, and D. B. Skillicorn. Predictable Communication on Unpredictable Networks: Implementing BSP over TCP/IP. In *Proc. 4th Intl. Euro-Par Conf.*, number 1470 in LNCS, pages 970–980, Southampton, UK, September 1998.

[69] J. J. Dongarra. Performance of Various Computers Using Standard Equations Software in a Fortran Environment. *Computer Architecture News*, 18(2):17–31, 1990.

[70] J. J. Dongarra, S. W. Otto, M. Snir, and D. Walker. A Message Passing Standard for MPP and Workstations. *Comm. of the ACM*, 39(7):84–90, 1996.

[71] P. Druschel. Operating System Support for High-Speed Communication. *Comm. of the ACM*, 39(9):41–51, 1996.

[72] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li. VMMC-2: Efficient Support for Reliable, Connection-Oriented Communication. In *Proc. Hot Interconnects V*, Palo Alto, CA, August 1997. `http://www.cs.princeton.edu/shrimp/Papers/hotIC97VMMC2.ps`. Also as TR 573-98, Department of Computer Science, Princeton University.

[73] A. C. Dusseau, R. H. Arpaci, and D. E. Culler. Effective Distributed Scheduling of Parallel Workloads. In *Proc. SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 25–36, Philadelphia, PA, May 1996. `http://HTTP.CS.Berkeley.EDU/~dusseau/Papers/sigmetrics96.ps`.

[74] H. El-Rewinig, H. H. Ali, and T. Lewis. Task Scheduling in Multiprocessing Systems. *IEEE Computer*, 28(12): 27–37, 1995.

[75] E. N. Elnozahy, D. B. Johnson, and Y. M. Wang. A Survey of Rollback-Recovery Protocols in Message Passing Systems. Technical Report CMU-CS-96-181, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, September 1996.

[76] D. G. Feitelson and L. Rudolph. Gang Scheduling Performance Benefits for Fine-Grain Synchronization. *J. of Parallel and Distributed Computing*, 16(4):306–318, 1992.

[77] A. J. Ferrari. JPVM: Network Parallel Computing in Java. In *Proc. ACM Workshop on Java for High-Performance Network Computing*, pages 245–250, Palo Alto, CA, February 1998.

[78] S. M. Figueira and F. Berman. Predicting Slowdown for Networked Workstations. Technical Report CS97-525, Department of Computer Science, University of California, San Diego, CA, February 1997. `http://www.cs.ucsd.edu/groups/hpcl/apples/pubs/silvia4.ps`.

[79] D. Flanagan. *JavaScript The Definitive Guide*. O'Reilly & Associates, Inc., Sebastopol, CA, 1997.

[80] G. Fohler. Realizing Changes of Operational Modes with a Pre-Run-Time Scheduled Hard Real-Time System. In *Proc. 2nd Workshop on Responsive Computer Systems*, pages 122–128, Saitama, Japan, 1992.

[81] S. Fortune and J. Wyllie. Parallelism in Random Access Machines. In *Conf. Record 10th Symp. on Theory of Computing*, pages 114–118, San Diego, CA, May 1978.

[82] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *Intl. J. of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.

[83] H. Franke, P. Pattnaik, and L. Rudolph. Gang Scheduling for Highly Efficient Distributed Multiprocessor Systems. In *Proc. 6th Symp. on the Frontiers of Massively Parallel Computation*, pages 4–12, Annapolis, MD, October 1996.

[84] B. Furht. Parallel Computing: Glory and Collapse. *IEEE Computer*, 27(11):74–75, 1994.

[85] R. Gamache, R. Short, and M. Massa. Windows NT Clustering Service. *IEEE Computer*, 31(10):55–62, 1998.

[86] R. Geist, R. Reynolds, and J. Westall. Selection of a Checkpoint Interval in a Critical-Task Environment. *IEEE Trans. on Reliability*, 37(4):395–400, 1988.

[87] Gigabit Ethernet Accelerating the Standard for Speed — Whitepaper. Unpublished manuscript, `http://www.gigabit-ethernet.org/technology/whitepapers/gige_97/gigabit2.pdf`, 1997.

[88] R. Gillet. Memory Channel Network for PCI. *IEEE Micro*, 16(1):12–18, 1996. `http://www.digital.com/info/hpc/ref/ieee.html`.

[89] The Globus Resource Specification Language RSL v1.0. Language manual at `http://www-fp.globus.org/gram/rsl_spec1.html`, April 1999.

[90] R. Gopalakrishnan and G. M. Parulkar. A Framework for QoS Guarantees for Multimedia Applications within an Endsystem. In *Proc. Herausforderungen eines globalen Informationsverbundes für die Informatik, GI Jahrestagung*, pages 43–56, Zürich, Switzerland, September 1995.

[91] R. Gopalakrishnan and G. M. Parulkar. Bringing Real-time Scheduling Theory and Practice Closer for Multimedia Computing. In *Proc. SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 1–12, Philadelphia, PA, May 1996.

[92] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, Inc., Reading, MA, 1996.

[93] M. D. Grammatikakis, D. F. Hsu, M. Kraetzl, and J. F. Sibeyn. Packet Routing in Fixed-Connection Networks: A Survey. *J. of Parallel and Distributed Computing*, 54(2):77–132, 1998.

[94] V. Grassi, L. Donatiello, and S. Tucci. On the Optimal Checkpointing of Critical Tasks and Transaction-Oriented Systems. *IEEE Trans. on Software Engineering*, 18(1):72–77, 1992.

[95] J. Gray and D. Siewiorek. High-Availability Computer Systems. *IEEE Computer*, 24(9):39–48, 1991.

[96] A. Grimshaw, A. Ferrari, G. Lindahl, and K. Holcomb. Metasystems. *Comm. of the ACM*, 41(11):46–55, 1998.

[97] A. S. Grimshaw and W. A. Wulf. The Legion Vision of a Worldwide Virtual Computer. *Comm. of the ACM*, 40 (1):39–45, 1997.

[98] R. Guerraoui and A. Schiper. Software-Based Replication for Fault Tolerance. *IEEE Computer*, 30(4):68–74, 1997.

[99] V. Hadzilacos and S. Toueg. *Distributed Systems*, chapter Fault-Tolerant Broadcasts and Related Problems, pages 97–145. In MULLENDER [206], 2nd edition, 1993.

[100] S. Han and K. Shin. A Primary-Backup Channel Approach to Dependable Real-Time Communication in Multihop Networks. *IEEE Trans. on Computes*, 47(1):46–61, 1998.

[101] R. Harper and B. Flahive. Challenges for Continuously Available Systems. *IEEE Computer*, 31(4):39–40, 1998.

[102] T. J. Harris. A Survey of PRAM Simulation Techniques. *ACM Computing Surveys*, 26(2):187–206, 1994.

[103] J. Hartung, B. Elpelt, and K.-H. Klössner. *Statistik*. R. Oldenbourg, München, 1993.

[104] J. M. D. Hill, S. Donaldson, and D. Skillicorn. Stability of Communication Performance in Practice: from the Cray T3E to Networks of Workstations. Technical Report PRG-TR-33-97, Programming Research Group, Oxford University Computing Laboratory, October 1997.

[105] M. Hill. Multiprocessors Should Support Simple Memory-Consistency Models. *IEEE Computer*, 31(8):28–34, 1998.

[106] W. D. Hillis. *The Connection Machine*. MIT Press, Cambride, MA, 1985.

[107] C. A. R. Hoare. Communicating Sequential Processes. *Comm. of the ACM*, 21(8):666–677, 1978.

[108] A. Hori, F. B. O'Carroll, H. Tezuka, and Y. Ishikawa. Gang Scheduling vs. Coscheduling: A Comparison with Data-Parallel Workloads. In *Proc. Intl. Conf. on Parallel and Distributed Processing Techniques and Applications*, volume 2, pages 1050–1057, Las Vegas, NV, July 1998.

[109] R. W. Horst. TNet: A Reliable System Area Network. *IEEE Mirco*, 15(1):37–45, 1995.

[110] D. Hyde. TFCC Newsletter—Dialogs, 1(1). Web publication at `http://www.eg.bucknell.edu/~hyde/tfcc/vol1no1-dialog.html`, April 1999.

[111] IEEE Computer Scociety: Parascope. `http://computer.org/parascope`, August 1998.

[112] IEEE Standard 1596-1992 for Scalable Coherent Interface (SCI), 1992. `http://standards.ieee.org/catalog/`.

[113] ISO. ISO/IEC 9945-1. Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C Language]. ANSI/IEEE Std 1003.1, 2nd edition, 1996-07-12, 1996.

[114] QoS—Basic Framework. CD Text ISO/IEC JTC 1/SC 21 N9309, January 1995.

[115] ITU-T. Quality of Service, Network Management and Traffic Engineering — Terms and Definitions Related to Quality of Service and Network Performance Including Dependability. ITU-T Recommendation E.800, November 1994.

[116] ITU. Information Technology—Open Distributed Processing—Reference Model. ITU Standard X.902, 1995.

[117] K. K. Jain and V. Rajaraman. Lower and Upper Bounds on Time for Multiprocessor Optimal Schedules. *IEEE Trans. on Parallel and Distributed Systems*, 5(8):879–886, 1994.

[118] K. Jeong, D. Shasha, S. Talla, and P. Wyckoff. An Approach to Fault-Tolerant Parallel Processing on Intermittently Idle, Heterogeneous Workstations. In *Proc. 27th Intl. Symp. on Fault-Tolerant Computing*, pages 11–20, Seattle, WA, June 1997.

[119] D. Jewett. Integrity S2 — A Fault-Tolerant UNIX Platform. In *Proc. 21th Intl. Symp. on Fault-Tolerant Computing*, pages 512–519, Montreal, CA, June 1991. Reprinted in [259].

[120] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proc. 15th Symp. on Operating Systems Principles*, pages 213–228, Cooper Mountain Resort, CO, March 1995. `ftp://cag.lcs.mit.edu/pub/papers/crl-sosp-15.ps.Z`.

[121] M. B. Jones, J. S. Barrer, A. Forin, P. J. Leach, D. Rosu, and M.-C. Rosu. An Overview of the Rialto Real-Time Architecture. In *Proc. 7th SIGOPS European Workshop: Systems Support for Worldwide Applications*, pages 249–256, Connemara, Ireland, September 1996. `http://research.microsoft.com/~mbj/papers/tr-96-13.ps`.

[122] M. B. Jones, P. J. Leach, R. P. Draves, and J. S. Barrera. Modular Real-Time Resource Management in the Rialto Operating System. In *Proc. 5th Workshop on Hot Topics in Operating Systems*, pages 12–17, Orcas Island, WA, May 1995. `http://research.microsoft.com/~mbj/papers/tr-95-16.ps`.

[123] M. B. Jones and J. Regehr. Issues in Using Commodity Operating Systems for Time-Dependent Tasks: Experiences from a Study of Windows NT. In *Proc. 8th Intl. Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 107–110, Cambridge, England, July 1998. `http://research.microsoft.com/~mbj/papers/tr-98-29.ps`.

[124] M. F. Kaashoek and A. S. Tanenbaum. Group Communication in the Amoeba Distributed Operating System. In *Proc. 11th Intl. Conf. on Distributed Computing Systems*, pages 222–230, Arlington, TX, May 1991.

[125] J. Kamada, M. Yuharo, and E. Ono. User-level Realtime Scheduler Exploiting Kernel-level Fixed Priority Scheduler. In *Proc. Intl. Symp. on Multimedia Systems*, Yokohama, Japan, March 1996.

[126] D. D. Kandlur, K. G. Shin, and D. Ferrari. Real-Time Communication in Multihop Networks. *IEEE Trans. on Parallel and Distributed Systems*, 5(10):1044–1056, 1994.

[127] A. Kanevsky, A. Skjellum, and A. Rounbehler. MPI/RT — An Emerging Standard for High-Performance Real-Time Systems. In *Proc. 31st Hawaii Intl. Conf. on System Sciences*, volume 3 (Emerging Technologies Track), pages 157–166, Kohala Coast, HI, January 1998. `http://www.mpirt.org/documents/hicss31_paper.pdf`.

[128] A. Kanevsky, A. Skjellum, and J. Watts. Standardization of Communication Middleware for High-Performance Real-Time Systems. In *Proc. Workshop on Middleware for Distributed Real-Time Systems and Services*, pages 206–213, San Francisco, CA, December 1997. `http://www.mpirt.org/documents/rtss97_paper.pdf`.

[129] C. T. Karamanolis and J. N. Magee. Client-Access Protocols for Replicated Services. *IEEE Trans. on Software Engineering*, 25(1):3–21, 1999.

[130] H. Karl. Bridging the Gap between Distributed Shared Memory and Message Passing. *Concurrency: Practice and Experience*, 10(11–13):887–900, 1998.

[131] H. Karl. Bridging the Gap between Distributed Shared Memory and Message Passing. In *Proc. Workshop on Java for High-Performance Network Computing*, pages 171–180, Palo Alto, CA, February 1998.

[132] H. Karl. Coscheduling through Synchronized Scheduling Servers — A Prototype and Experiments. Informatik-Berichte 112, Institut für Informatik, Humboldt-Universität, Berlin, Germany, August 1998.

[133] H. Karl. Coscheduling through Synchronized Scheduling Servers—A Prototype and Experiments. In *Parallel and Distributed Processing*, number 1586 in LNCS, pages 1115–1129, San Jose, Puerto Rico, April 1999. Proc. of Workshop on Personal Computer Based Networks of Workstations, held in conjunction with IPPS/SPDP '99.

[134] H. Karl. A Rigorous Analysis of Eager Scheduling's Run Time Distribution. Informatik-Berichte 121, Institut für Informatik, Humboldt-Universität, Berlin, Germany, April 1999.

[135] H. Karl, A. Polze, and M. Werner. Predictable Network Computing using Message-Driven Scheduling. In

Proc. Workshop on Run-Time Systems for Parallel Programming, number IR-417 in Tech. Report of the Vrije Universitaet Amsterdam, pages 1–10, April 1997.

[136] H. Karl, A. Polze, and M. Werner. A Flexible, Distributed I/O-based Framework for Improved Fault Tolerance of Legacy Software. Informatik-Berichte 118, Institut für Informatik, Humboldt-Universität, Berlin, Germany, February 1999.

[137] H. Karl and M. Werner. An Experimental Investigation of Message Latencies in the Totem Protocol in the Presence of Faults. Informatik-Berichte 93, Institut für Informatik, Humboldt-Universität, Berlin, Germany, 1997.

[138] H. Karl and M. Werner. An Optimal Checkpointing Interval for Real-Time Systems. In *Proc. Intl. Conf. Parallel and Distributed Processing Techniques and Applications*, pages 604–612, Las Vegas, NV, July 1997.

[139] H. Karl and M. Werner. An Optimal Checkpointing Interval for Real-Time Systems. Informatik-Berichte 92, Institut für Informatik, Humboldt-Universität, Berlin, Germany, December 1997.

[140] H. Karl, M. Werner, and L. Küttner. An Experimental Investigation of Message Latencies in the Totem Protocol in the Presence of Faults. *IEE Software*, 145(6):219–227, 1998.

[141] H. Karl, M. Werner, and L. Küttner. An Experimental Investigation of Message Latencies in the Totem Protocol in the Presence of Faults. In *Proc. Euromicro*, pages 468–475, Västeras, Sweden, August 1998.

[142] D. O. Keck and P. J. Kuehn. The Feature and Service Interaction Problem in Telecommunications Systems: A Survey. *IEEE Trans. on Software Engineering*, 24(10):779–796, 1998.

[143] A.-M. Kermarrec, G. Cabillic, A. Gefflaut, C. Morin, and I. Puaut. A Recoverable Distributed Shared Memory Integrating Coherence and Recoverability. In *Proc. 25th Intl. Symp. on Fault-Tolerant Computing*, pages 289–298, Pasadena, CA, June 1995.

[144] Y. A. Khalidi, J. M. Bernabeu, V. Matena, K. Shirriff, and M. Thadani. Solaris MC: A Multi-Computer OS. Technical Report TR-95-48, Sun Microsystems Laboratories, Palo Alto, CA, November 1995. `http://www.sunlabs.com/techrep/1995/smli_tr-95-48.pdf`.

[145] R. M. Kieckhafer, C. J. Walter, A. M. Finn, and P. M. Thambidurai. The MAFT Architecture for Distributed Fault-Tolerant Computing. *IEEE Trans. on Computers*, 37(4):398–405, 1988.

[146] J. Kim and D. Lilja. A Network Status Predictor to Support Dynamic Scheduling in Network-based Computing Systems. In *Proc. 13th Intl. Parallel Processing Symp./10th Symp. on Parallel and Distributed Processing*, pages 372–378, San Juan, Puerto Rico, April 1999.

[147] S. Kim and K. Park. Fully-Scalable Fault-Tolerant Simulations for BSP and CGM. In *Proc. 13th Intl. Parallel Processing Symp./10th Symp. on Parallel and Distributed Processing*, pages 117–124, San Juan, Puerto Rico, April 1999.

[148] H. Kopetz. The Failure Fault (FF) Model. In *Proc. 12th Intl. Symp. on Fault-Tolerant Computing*, pages 14–18, Santa Monica, CA, June 1982.

[149] H. Kopetz. *Real-Time Systems Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1997.

[150] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. Distributed Fault-Tolerant Real-Time Systems: The Mars Approach. *IEEE Micro*, 9(1):25–40, 1989.

[151] H. Kopetz and G. Grünsteidl. TTP—A Protocol for Fault-Tolerant Real-Time Systems. *IEEE Computer*, 27(1): 14–23, 1994.

[152] H. Kopetz and P. Verissimo. *Distributed Systems*, chapter Real Time and Dependability Concepts, pages 411–446. In MULLENDER [206], 2nd edition, 1993.

[153] C. M. Krishna and K. G. Shin. *Real-Time Systems*. McGraw Hill, New York, NY, 1997.

[154] C.M. Krishna, K.G. Shin, and Y.-H. Lee. Optimization Criteria for Checkpointing. *Comm. of the ACM*, 27(10): 1008–1012, 1984.

[155] J. Kurose. Open Issues and Challenges in Providing Quality of Service Guarantees in High-Speed Networks. *ACM Comp. Comm. Review*, 23(1):6–15, 1993.

[156] L. Küttner. Gruppenkommunikation für Responsivität. Diploma thesis, Institut für Informatik, Humboldt-Universität zu Berlin, Germany, July 1997.

[157] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Trans. on Computer*, C-28(9):690–691, 1979.

[158] L. Lamport and P. M. Melliar-Smith. Synchronizing Clocks in the Presence of Faults. *J. of the ACM*, 32(1):52–78, 1989.

[159] J.-C. Laprie. Dependable Computing and Fault Tolerance: Concepts and Terminology. In *Proc. 15th Intl. Symp. on Fault-Tolerant Computing*, pages 2–11, Ann Arbor, MI, June 1985. Reprinted in [259].

[160] J. C. Laprie, editor. *Dependability: Basic Concepts and Terminology*, volume 5 of *Dependable Computing and Fault-Tolerant Systems*. Springer-Verlag, 1991.

[161] L. Laranjeira, M. Malek, and R. Jenevein. On Tolerating Faults in Naturally Redundant Algorithms. In *Proc. 10th Symp. on Reliable Distributed Systems*, pages 118–127, Pisa, Italy, September 1991.

[162] L. A. Laranjeira. NCAPS: Application High Availability in UNIX Computer Clusters. In *Proc. 28th Intl. Symp. on Fault-Tolerant Computing*, pages 441–450, Munich, Germany, June 1998.

[163] L. A. Laranjeira, M. Malek, and R. Jenevein. Nest: A Nested-Predicate Scheme for Fault Tolerance. *IEEE Trans. on Computers*, 42(11):1303–1324, 1993.

[164] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. *Computer Architecture News*, 25(2):241–251, 1997. `http://www.sgi.com/Technology/CompCon/isca.pdf`.

[165] Jean-Yves Le Boudec. The Asynchronous Transfer Mode: A Tutorial. *Computer Networks and ISDN Systems*, 24:279–309, 1992.

[166] C. Lee, C. Kesselman, and S. Schwab. Near-real-time Satellite Image Processing. *IEEE Computer Graphics and Applications*, 16:79–84, July 1996. `ftp://ftp.globus.org/pub/globus/papers/neph.pdf`.

[167] C. Lee, C. Kesselman, J. Stepanek, R. Lindell, S. Hwang, B. Scott Michel, J. Bannister, I. Foster, and A. Roy. Qualis: the Quality of Service Component for the Globus Metacomputing System. In *Proc. 6th Intl. Workshop on Quality of Service*, pages 140–142, Napa, CA, May 1998. `ftp://ftp.globus.org/pub/globus/papers/qualis_pp.pdf`.

[168] C. Lee, R. Rajkumar, and C. Mercer. Experiences with Processor Reservation and Dynamic QOS in Real-Time Mach. In *Proc. Intl. Symp. on Multimedia Systems*, Yokohama, Japan, April 1996. `http://www.cs.cmu.edu/afs/cs/project/rtmach/public/papers/mm-japan-96.ps`.

[169] C. Lee, K. Yoshida, C. Mercer, and R. Rajkumar. Predictable Communication Protocol Processing in Real-Time Mach. In *Proc. 2nd Real-Time Technology and Applications Symp.*, pages 220–229, Boston, MA, June 1996. `http://www.cs.cmu.edu/afs/cs/project/rtmach/public/papers/rtas96.ps`.

[170] J. P. Lehoczky. Real-Time Queueing Theory. In *Proc. 17th Real-Time Systems Symp.*, pages 186–195, Washington, D.C., December 1996.

[171] J. P. Lehoczky, L. Sha, and J. K. Strosnider. Enhanced Aperiodic Responsiveness in Hard Real-Time Environments. In *Proc. Real-Time Systems Symp.*, pages 261–270, December 1987.

[172] W. E. Leland, M. S. Taqq, W. Willinger, and D. V. Wilson. On the Self-Similar Nature of Ethernet Traffic. In D. P. Sidhu, editor, *ACM SIGCOMM*, pages 183–193, San Francisco, California, September 1993. `ftp://thumper.bellcore.com/pub/world/wel/sigcomm93.ps.Z`.

[173] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash Multiprocessor. *IEEE Computer*, 25(3):63–79, 1992.

[174] P. Leydekkers, V. Gay, and L. Franken. A Computational and Engineering View on Open Distributed Real-Time Multimedia Exchange. In *Network and Operating System Support for Digital Audio and Video, Proc. 5th International Workshop, NOSSDAV '95*, number 1018 in LNCS, pages 41–52, Durham, NH, April 1995.

[175] K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proc. Intl. Conf. on Parallel Processing, Vol. 2: Software*, pages 94–101, University Park, PA, August 1988.

[176] K. Li. Stochastic Bounds for Parallel Execution Times with Processor Constraints. *IEEE Trans. on Computers*, 465:630–636, May 1997.

[177] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. on Computer Systems*, 7(4):321–359, 1989.

[178] K. Li and X. Sun. Average-Case Analysis of Isospeed Scalability of Parallel Computations on Multiprocessors. In *Proc. 13th Intl. Parallel Processing Symp./10th Symp. on Parallel and Distributed Processing*, pages 112–116, San Juan, Puerto Rico, April 1999.

[179] D. Libes. Expect: Curing Those Uncontrollable Fits of Interaction. In *Proc. Summer 1990 USENIX Conf.*, pages 183–192, Anaheim, CA, June 1990.

[180] C. Lin, H. Chu, and K. Nahrstedt. A Soft Real-time Scheduling Server on the Windows NT. In *2nd USENIX Windows NT Symp.*, pages 149–155, Seattle, WA, August 1998. `http://cairo.cs.uiuc.edu/papers/usenix-nt-98.ps`.

[181] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor : A Hunter of Idle Workstations. In *Proc. 8th Intl. Conf. on Distributed Computing Systems*, pages 104–111, Washington, D.C., June 1988.

[182] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. of the ACM*, 20(1):46–61, 1973.

[183] J. W. S. Liu, W.-K. Shih, K.-W. Lin, R. Bettati, and J.-Y. Chung. Imprecise Computations. *Proc. of the IEEE*, 82 (1):83–94, 1994.

[184] H. Lu, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Message Passing Versus Distributed Shared Memory on Networks of Workstations. In *Proc. Supercomputing*, San Diego, CA, December 1995. `http://www.cs.rice.edu/~willy/papers/sc95.ps.gz` or `http://www.supercomp.org/sc95/proceedings/586_HLU/SC95.HTM`.

[185] P. Lu. Aurora: Scoped Behavior for Per-Context Optimized Distributed Data Sharing. In *Proc. 11th Intl. Parallel Processing Symp.*, pages 467–473, Geneva, Switzerland, April 1997.

[186] S.-W. Luan and V. D. Gligor. A Fault-Tolerant Protocol for Atomic Broadcast. *IEEE Trans. on Parallel and Distributed Systems*, 1(3):271–285, 1990.

[187] S. Madala and J. B. Sinclair. Performance of Synchronous Parallel Algorithms with Regular Structures. *IEEE Trans. on Parallel and Distributed Systems*, 2(1):105–116, 1991.

[188] M. Malek. Responsive Systems (The Challenge for the Nineties). *Microprocessing & Microprogramming*, 30: 9–16, 1990.

[189] M. Malek. Responsive Systems: A Marriage between Real Time and Fault Tolerance. In *Proc. 5th Intl. Conf. on Fault-Tolerant Computing Systems*, number 283 in Informatik-Fachbericht, pages 1–17. Springer, 1991.

[190] M. Malek. A Consensus-Based Framework and Model for the Design of Responsive Computing Systems. In G. M. Koob and C.G. Lau, editors, *Foundations of Dependable Computing: Models and Frameworks for Dependable Systems*, chapter 1.1, pages 3–21. Kluwer Academic Publishers, Boston, MA, 1994.

[191] M. Malek. Omniscience, Consensus, Autonomy: Three Tempting Roads to Responsiveness. In *Proc. 14th Symp. on Reliable Distributed Systems*, pages xii–xiv, Bad Neuenahr, Germany, September 1995.

[192] M. Malek, A. Polze, and M. Werner. A Framework for Responsive Parallel Computing in Network-based Systems. In *Proc. Intl. Workshop on Advanced Parallel Processing Technologies*, pages 335–343, Bejing, China, September 1995.

[193] D. Martinez. Application of Parallel Processors to Real-Time Sensor Array Processing. In *Proc. 13th Intl. Parallel Processing Symp./10th Symp. on Parallel and Distributed Processing*, pages 463–469, San Juan, Puerto Rico, April 1999.

[194] G. M. Masson, D. M. Blough, and G. F. Sullivan. *Fault-tolerant Computer System Design*, chapter System Diagnosis, pages 478–536. In PRADHAN [231], 1996.

[195] O. A. McBryan. An Overview of Message Passing Environments. *Parallel Computing*, 20(4):417–444, 1994.

[196] A. Mehra, A. Indiresan, and K. Shin. Resource Management for Real-Time Communication: Making Theory Meet Practice. In *Proc. 2nd IEEE Real-Time Technology and Applications Symp.*, pages 130–138, Boston, MA, June 1996. `ftp://rtcl.eecs.umich.edu/outgoing/ashish/rtas96-final.ps.Z`.

[197] A. Mehra, A. Indiresan, and K. G. Shin. Structuring Communication Software for Quality-of-Service Guarantees. In *Proc. Real-Time Systems Symp.*, pages 144–154, Washington, D.C., December 1996.

[198] C. W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves: Operating System Support for Multimedia Applications. In *Proc. Intl. Conf. on Multimedia Computing and Systems*, pages 90–99, Boston, MA, May 1994. `http://www.cs.cmu.edu/afs/cs/project/rtmach/public/papers/mcs94.ps`.

[199] Message Passing Interface Forum. MPI: A Message Passing Interface Standard, May 1994. `www.mpi-forum.org/docs/mpi-10.ps`.

[200] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface, May 1997. `www.mpi-forum.org/docs/mpi-20.ps`.

[201] J. F. Meyer. On Evaluating the Performability of Degradable Computing Systems. *IEEE Trans. on Computers*, C-29(8):720–731, 1980.

[202] J. F. Meyer. Performability: A Retrospective and Some Pointers to the Future. *Performance Evaluation*, 14: 139–156, February 1992.

[203] L. E. Moser and P. M. Melliar-Smith. Probabilistic Bounds on Message Delivery for the Totem Single-Ring Protocol. In *Proc. 15th Real-Time Systems Symp.*, pages 238–248, San Juan, Puerto Rico, December 1994.

[204] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: A Fault-Tolerant Multicast Group Communication System. *Comm. of the ACM*, 39(4):54–63, 1996.

[205] S. Mukherjee and M. Hill. Making Network Interfaces Less Peripheral. *IEEE Computer*, 31(10):70–76, 1998.

[206] S. Mullender, editor. *Distributed Systems*. ACM Press, New York, NY, 2nd edition, 1993.

[207] S. P. Muppala, J. K. Woolet and K. S. Trivedi. Real-Time-Systems Performance in the Presence of Failures. *IEEE Computer*, 24(5):37–47, 1991.

[208] K. Nahrstedt. End-to-End QoS Guarantees in Networked Multimedia Systems. *ACM Computing Surveys*, 27(4): 613–616, 1995.

[209] K. Nahrstedt and J. Smith. End-to-End QoS Guarantees: Lessons Learned from OMEGA. Technical Report UIUCDCS-R-96-1957, Department of Computer Science, University of Illinois, Urbana-Champaign, IL, July 1996.

[210] K. Nahrstedt and J. M. Smith. An Experimental Investigation of Issues in End-to-End QoS. Technical Report MS-CIS-94-08, University of Pennsylvania, February 1994.

[211] K. Nahrstedt and J. M. Smith. The QoS Broker. *IEEE Multimedia Magazine*, 2(1):53–67, 1995.

[212] R. S. Nikhil. Cid: A Parallel "Shared-Memory" C for Distributed Memory Machines. In *Proc. 7th Ann. Workshop on Languages and Compilers for Parallel Computing*, volume 892 of *LNCS*, pages 376–390, Ithaca, NY, August 1994.

[213] R. S. Nikhil, G. M. Papadopoulous, and Arvind. *T: A Multithreaded Massively Parallel Architecture. In *Proc. 19th Intl. Symp. on Computer Architecture*, pages 156–167, Gold Coast, Australia, May 1992. `ftp://csg-ftp.lcs.mit.edu/pub/papers/csgmemo/memo-325-1.ps.Z`.

[214] K. Nilsen. Adding Real-Time Capabilities to Java. *Comm. of the ACM*, 41(6):49–56, 1998.

[215] B. Nitzberg and V. Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *IEEE Computer*, 24(8): 52–60, 1991.

[216] Object Management Group. The Common Object Request Broker: Architecture and Specification, 1995. Object Management Group, Inc., Framingham, MA, USA.

[217] Object Management Group. Fault tolerant CORBA Using Entity Redundancy RFP. `ftp://ftp.omg.org/pub/docs/orbos/98-04-01.pdf`, April 1998. Object Management Group, Inc., Framingham, MA, USA.

[218] Object Management Group. Realtime Technologies, Request for Information. `ftp://ftp.omg.org/pub/docs/orbos/1996/96-09-02.pdf`, 1996. Object Managment Group, Inc., Framingham, MA, USA.

[219] J. K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proc. 3rd Intl. Conf. on Distributed Computing Systems*, pages 22–30, Miami, FL, October 1982.

[220] S. Pakin, V. Karamcheti, and A. Chien. Fast Messages: Efficient Portable Communication for Workstation Clusters and MPPs. *IEEE Concurrency*, 5(2):60–73, 1997. `http://www-csag.cs.ucsd.edu/papers/csag/external/fm-pdt.ps`.

[221] V. Paxson, G. Almes, J. Mahdavi, and M. Mathis. Framework for IP Performance Metrics. RFC 2330, May 1998.

[222] V. Paxson and S. Floyd. Wide-Area Traffic: The Failure of Poisson Modeling. *IEEE/ACM Transaction on Networking*, 3(3):226–244, 1995.

[223] G. Pfister. *In Search of Clusters: The Coming Battle in Lowly Parallel Computing*. Prentice Hall, Inc., Englewood Cliffs, NJ, 1995.

[224] M. Philippsen and M. Zenger. JavaParty—Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, 1997.

[225] J. S. Plank and W. R. Elwasif. Experimental Assessment of Workstation Failures and their Impact on Checkpointing Systems. In *Proc. 28th Intl. Symp. on Fault-Tolerant Computing*, pages 48–57, Munich, Germany, June 1998.

[226] A. Polze. How to Partition a Workstation. In *Proc. 8th IASTED/ISMM Intl. Conf. on Parallel and Distributed Computing and Systems*, pages 184–187, Chicago, IL, October 1996. `http://www.informatik.hu-berlin.de/~apolze/papers/pdcs96.ps`.

[227] A. Polze and M. Malek. Parallel Computing in a World of Workstations. In *Proc. 7th IASTED/ISMM Intl. Conf. on Parallel and Distributed Computing and Systems*, pages 72–75, Washington, D.C., October 1995. `http://www.informatik.hu-berlin.de/~apolze/papers/pdcs95.ps`.

[228] A. Polze, M. Werner, and M. Malek. High-Performance Responsive Computing with CORE and SONiC. Informatik-Berichte 75, Humboldt-Universität, Berlin, Germany, December 1996.

[229] D. Powell. Failure Mode Assumptions and Assumption Coverage. Technical Report 91462, Laboratoire d'Analyse et d'Architecture des Systemes (LAAS), Toulouse, France, 1995. `http://doc.laas.fr:8889/`.

[230] D. Powell, P. Verissimo, G. Bonn, F. Waselznck, and D. Seaton. The Delta-4 Approach to Dependability in Open Distributed Computing Systems. In *Proc. 18th Intl. Symp. on Fault-Tolerant Computing*, pages 246–251, Tokyo,

Japan, June 1988. Reprinted in [259].

[231] D. K. Pradhan, editor. *Fault-tolerant Computer System Design*. Prentice Hall, Upper Saddle River, NJ, 1996.

[232] F. Preparata, G. Metze, and R. Chien. On the Connection Assignment Problem of Diagnosable Systems. *IEEE Trans. on Computers*, C-16:848–854, December 1967.

[233] J. Protić, M. Tomašević, and V. Milutinović, editors. *Distributed Shared Memory: Concepts and Systems*. IEEE Computer Society Press, Los Alamitos, CA, 1998.

[234] P. Puschner and C. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *J. Real-Time Systems*, 1(2):159–176, 1989.

[235] R. Rajkumar and M. Gagliardi. High Availability in the Real-Time Publisher/Subscriber Inter-Process Communication Model. In *Proc. Real-Time Systems Symposium*, pages 136–141, Washington, D.C., December 1996. `http://www.cs.cmu.edu/afs/cs/project/rtmach/public/papers/pub-sub-rejoin.ps`.

[236] R. Rajkumar, M. Gagliardi, and L. Sha. The Real-Time Publisher/Subscriber Inter-Process Communication Model for Distributed Real-Time Systems: Design and Implementation. In *Proc. 1st Real-Time Technology and Applications Symposium*, pages 66–75, Chicago, IL, May 1995. `http://www.cs.cmu.edu/afs/cs/project/rtmach/public/papers/pub-sub.ps`.

[237] W. Rehm and A. Munke. Cluster Computing. `http://www.tu-chemnitz.de/informatik/RA/cchp/index.html`, April 1999.

[238] J. Rexford, J. Hall, and K. G. Shin. A Router Architecture for Real-Time Communication in Multicomputer Networks. *IEEE Trans. on Computers*, 47(10):1088–1101, 1998.

[239] J. Richling and A. Polze. Scheduling Server for Predictable Computing: An Experimental Evaluation. In *Proc. Workshop on Middleware for Distributed Real-Time Systems and Services*, pages 130–137, San Francisco, CA, December 1997.

[240] D. Ridge, D. Becker, P. Merkey, and T. Sterling. Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs. In *Proc. IEEE Aerospace*, 1997. `http://beowulf.gsfc.nasa.gov/papers/AA97/aa97.ps`.

[241] M. C. Rinard, D. J. Scales, and M. S. Lam. Heterogeneous Parallel Programming in Jade. In *Proc. Supercomputing*, pages 245–256, Minneapolis, MN, November 1992. `http://www-suif.stanford.edu/papers/`.

[242] M. C. Rinard, D. J. Scales, and M. S. Lam. Jade: A High-Level, Machine-Independent Language for Parallel Programming. *IEEE Computer*, 26(6):28–38, 1993.

[243] L. Sachs. *Angewandte Statistik*. Springer, Berlin, 7th edition, 1992.

[244] J. Saltz, A. Sussman, S. Graham, J. Demmel, S. Baden, and J. Dongarra. Programming Tools and Environments. *Comm. of the ACM*, 41(11):64–73, 1998.

[245] S. Sardesi. *CHIME: A Versatile Distributed Parallel Processing System*. PhD thesis, Arizona State University, Tempe, AZ, May 1997.

[246] S. Sardesi, D. McLaughlin, and P. Dasgupta. Distributed Cactus Stacks: Runtime Stack-Sharing Support for Distributed Parallel Programs. In *Proc. Intl. Conf. Parallel and Distributed Processing Techniques and Applications*, pages 57–65, Las Vegas, NV, July 1997.

[247] A. Savva and T. Nanya. A Gracefully Degrading Massively Parallel System Using the BSP Model, and Its Evaluation. *IEEE Trans. on Computers*, 48(1):38–52, 1999.

[248] D. Scales and K. Gharacholoo. Towards Transparent and Efficient Software Distributed Shared Memory. In *Proc. 16th Symp. on Operating Systems Principles*, pages 157–169, St. Malo, France, October 1997.

[249] B. Schnor, S. Petri, and M. Becker. Scalability of Multicast Based Synchronization Methods. In *Proc. 1998 Euromicro Workshop on Parallel and Cluster Computing*, pages 969–975, Västeras, Sweden, August 1998.

[250] J. M. Schopf and F. Berman. Performance Prediction in Production Environments. In *Proc. 12th Intl. Parallel Processing Symp./9th Symp. on Parallel and Distributed Processing*, pages 647–653, Orlando, FL, March 1998. `http://www.cs.ucsd.edu/users/jenny/TechReports/ipps98.ps`. Also as TR CS97-558, Department of Computer Science and Engineering, University of California, San Diego, CA.

[251] S. Scott. Synchronization and Communication in the T3E Multiprocessor. In *Proc. 7th Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 26–36, Cambridge, MA, October 1996. `http://reality.sgi.com/sls_craypark/Papers/asplos96_t3e_comm.ps`.

[252] ServerNet. `http://www.servernet.com/`, January 1999. Company Webpage.

[253] L. Sha, R. Rajkumar, and M. Gagliardi. Evolving Dependable Real-Time Systems. In *Proc. IEEE Aerospace Applications Conf.*, pages 335–346, Aspen, CO, February 1996.

[254] L. Sha, S. S. Sathaye, and J. K. Strosnider. Analysis of Dual-Link Networks for Real-Time Applications. *IEEE Trans. on Computers*, 46(1):1–13, 1997.

[255] K. G. Shin, T.-H. Lin, and Y.-H. Lee. Optimal Checkpointing of Real–Time Tasks. *IEEE Trans. on Computer*, 36 (11):1328–1341, 1987.

[256] K. Shirriff. Building Distributed Process Management on an Object-Oriented Framework. In *Proc. USENIX Ann. Technical Conf.*, pages 119–131, Anaheim, CA, January 1997. `http://www.sunlabs.com/research/solaris-mc/doc/process-usenix.ps`.

[257] R. Short, R. Gamache, J. Vert, and M. Massa. Windows NT Clusters for Availability and Scalability. In *Proc. 42nd COMPCON*, San Jose, CA, February 1997.

[258] A. Siegel. Private communication, July 1998.

[259] D. P. Siewiorek, editor. *The Twenty-fifth International Symposium on Fault-Tolerant Computing—Highlights from 25 Years*. IEEE Computer Society Press, Los Alamitos, CA, 1995.

[260] D. P. Siewiorek and R. S. Swarz. *Reliable Computer Systems Design and Evaluation*. Digital Press, Burlington, MA, 2nd edition, 1992.

[261] L. Silva and J. Silva. The Performance of Coordinated and Independent Checkpointing. In *Proc. 13th Intl. Parallel Processing Symp./10th Symp. on Parallel and Distributed Processing*, pages 280–284, San Juan, Puerto Rico, April 1999.

[262] A. Singhai, S.-B. Lim, and S. R. Radia. The SunSCALR Framework for Internet Servers. In *Proc. 28th Symp. on Fault-Tolerant Computing*, pages 108–117, Munich, Germany, June 1998.

[263] L. Smarr and C. E. Catlett. Metacomputing. *Comm. of the ACM*, 35(6):45–52, 1993.

[264] W. Smith, I. Foster, and V. Taylor. Predicting Application Run Times Using Historical Information. In *Proc. 4th Workshop on Job Scheduling Strategies for Parallel Processing*, number 1459 in LNCS, pages 122–142, Orlando, FL, March 1998. `ftp://ftp.globus.org/pub/globus/papers/runtime.pdf`.

[265] P. G. Sobalvarro, S. Pakin, W. E. Weihl, and A. A. Chien. Dynamic Coscheduling on Workstation Clusters. Technical Report 1997-017, DEC Systems Research Center, Palo Alto, CA, March 1997. `ftp://ftp.digital.com/pub/DEC/SRC/technical-notes/SRC-1997-017.ps.gz`.

[266] A. H. Soukhanov, editor. *The American Heritage Dictionary of the English Language*. Houghton Mifflin, Boston, MA, 3rd edition, 1992.

[267] The Standard Performance Evaluation Cooperation. `http://www.spec.org`, May 1999.

[268] M. Spuri and G. Buttazzo. Scheduling Aperiodic Tasks in Dynamic Priority Systems. *Real-Time Systems*, 10(2): 179–210, 1996.

[269] B. Srinivasan. A Firm Real-Time System Implementation using Commercial Off-The-Shelf Hardware and Free Software. Master's thesis, Dept. of Electrical Engineering and Computer Science, University of Kansas, February 1998.

[270] J. Stankovic. Real-Time and Embedded Systems. *ACM Computing Surveys*, 28(1):205–208, 1996.

[271] J. A. Stankovic. Misconceptions About Real-Time Computing. *IEEE Computer*, 21(10):10–19, 1988.

[272] J. A. Stankovic and K. Ramamritham. What is Predictability for Real-Time Systems? *J. Real-Time Systems*, 2: 247–254, December 1990.

[273] J. A. Stankovic and K. Ramamritham. The Spring Kernel: A New Paradigm for Real-Time Systems. *IEEE Software*, 8(3):62–72, 1991.

[274] J. A. Stankovic and K. Ramamritham, editors. *Advances in Real-Time Systems*. IEEE Computer Society Press, Los Alamitos, CA, 1993.

[275] J. A. Stankovic and K. Ramamritham, editors. *Advances in Real-Time Systems*, chapter Programming Languages, pages 259–308. In STANKOVIC and RAMAMRITHAM [274], 1993.

[276] J. A. Stankovic, M. Spuri, M. Di Natale, and G. C. Buttazzo. Implications of Classical Scheduling Results for Real-Time Systems. *IEEE Computer*, 28(6):16–25, 1995.

[277] P. Steenkiste. Network-Based Multicomputers: A Practical Supercomputer Alternative. *IEEE Trans. on Parallel and Distributed Systems*, 7(8):830–840, 1996.

[278] R. W. Stevens. *Advanced Programming in the UNIX Environment*. Addison Wesley, Reading, MA, 1992.

[279] V. S. Sunderam, A. Geist, J. Dongarra, and R. Mancheck. The PVM Concurrent Computing System: Evolution, Experiences, and Trends. *Parallel Computing*, 20:531–546, April 1994.

[280] N. Suri, C. J. Walter, and M. M. Hugue, editors. *Advances in Ultra-Dependable Distributed Systems*. IEEE Computer Society Press, Los Alamitos, CA, 1995.

[281] B. N. Taylor. Guide for the Use of the International System of Units (SI). NIST Special Publication 811, 1995. `http://physics.nist.gov/Document/sp811.pdf`.

[282] D. Towsley. Providing Quality of Service in Packet Switched Networks. In L. Donatiello and R. Nelson, editors, *Performance Evaluation of Computer and Communications Systems*, pages 560–586. Springer Verlag, 1993.

[283] K. S. Trivedi. *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*. Prentice Hall, Englewood Cliffs, NJ, 1982.

[284] J. Turner. New Directions in Communications (or Which Way to the Information Age). *IEEE Communications Magazine*, 24(10):8–15, 1986.

[285] J. D. Ullman. NP-Complete Scheduling Problems. *J. of Computer and System Sciences*, 10:384–393, June 1975.

[286] N. H. Vaidya. Impact of Checkpoint Latency on Overhead Ratio of a Checkpointing Scheme. *IEEE Trans. on Computers*, 46(8):942–947, 1997.

[287] L. G. Valiant. A Bridging Model for Parallel Computation. *Comm. of the ACM*, 33(8):103–111, 1990.

[288] C.-J. L. van Driel, R. J. B. Follon, A. A. C. Kohler, R. P. M. van Osch, and J. M. Spanjers. The Error-Resistant Interactively Consistent Architecture (ERICA). In *Proc. 20th Intl. Symp. on Fault-Tolerant Computing*, pages 474–480, Newcastle upon Tyne, UK, June 1990. Reprinted in [280].

[289] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A Flexible Group Communication System. *Comm. of the ACM*, 39(4):76–83, 1996. `ftp://ftp.cs.cornell.edu/pub/isis/horus/doc/cacm.ps.Z`.

[290] R. J. Vetter. ATM Concepts, Architectures, and Protocols. *Comm. of the ACM*, 38(2):30–38, 1995.

[291] Virtual Interface Architecture Specification, Version 1.0. `http://www.viarch.org`, December 1997. (Registration is required.).

[292] W. Vogels, D. Dumitriu, K. Birman, R. Gamache, M. Massa, R. Short, J. Vert, J. Barrera, and J. Gray. The Design and Architecture of the Microsoft Cluster Service — A Practical Approach to High-Availability and Scalability. In *Proc. 28th Intl. Symp. on Fault-Tolerant Computing*, pages 422–431, Munich, Germany, June 1998.

[293] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proc. 15th Symp. on Operating System Principles*, pages 303–316, Copper Mountain, CO, December 1995.

[294] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. 19th Intl. Symp. on Computer Architecture*, pages 256–266, Gold Coast, Australia, May 1992.

[295] T. von Eicken and W. Vogels. Evolution of the Virtual Interface Architecture. *IEEE Computer*, 31(11):61–68, 1998.

[296] G. von Laszewski, M.-H. Su, J. A. Insley, I. Foster, J. Bresnahan, C. Kesselman, M. Thiebaux, M. L. Rivers, S. Wang, B. Tieman, and I. McNulty. Real-Time Analysis, Visualization, and Steering of Microtomography Experiments at Photon Sources. In *Proc. 9th SIAM Conf. on Parallel Processing for Scientific Computing*, San Antonio, TX, March 1999. `http://www.globus.org/documentation/incoming/siamCmt99.pdf`.

[297] B. Walker and D. Steel. Implementing a Full Single System Image UnixWare Cluster: Middleware vs. Underware. In *Technical Session on Cluster Computing Technologies, Environments, and Applications (CC-TEA) at PDPTA (to appear)*, Las Vegas, NV, June 1999. `http://www.dgs.monash.edu.au/~rajkumar/pdpta99/douglass.ps.gz`.

[298] Y.-M. Wang, Y. Huang, K.-P. Vo, P.-Y. Chung, and C. Kintala. Checkpointing and Its Applications. In *Proc. 25th Intl. Symp. on Fault-Tolerant Computing*, pages 22–31, Pasadena, CA, June 1995.

[299] C. B. Weinstock. SIFT: System Design and Implementation. In *Proc. 10th Intl. Symp. on Fault-Tolerant Computing*, pages 75–77, Kyoto, Japan, October 1980. Reprinted in [259].

[300] J. B. Weissman. Gallop: The Benefits of Wide-Area Computing for Parallel Processing. *J. of Parallel and Distributed Computing*, 54:183–205, November 1998.

[301] M. Welsh, A. Basu, and T. von Eicken. ATM and Fast Ethernet Network Interfaces for User-level Communication. In *Proc. 3rd Intl. Symp. on High Performance Computer Architecture*, pages 332–342, San Antonio, TX, February 1997.

[302] M. Werner. *Responsivität: Der konsensbasierte Ansatz*. PhD thesis, Humboldt-Universität, Berlin, Germany, 1999.

[303] M. Werner and H. Karl. Towards a Definition of Responsiveness. Informatik-Berichte 91, Institut für Informatik, Humboldt-Universität, Berlin, Germany, December 1997.

[304] M. Werner and M. Malek. The Unstoppables—Responsiveness by Consensus. Informatik-Berichte 90, Institut für Informatik, Humboldt-Universität, Berlin, Germany, 1997.

[305] M. Werner, A. Polze, and M. Malek. The Unstoppable Orchestra: A Responsive Distributed Application. In *Proc. 3rd Intl. Conf. on Configurable Distributed Systems*, pages 154–160, Annapolis, MD, May 1996.

[306] T. Wilkinson. Kaffe—A free virtual machine to run Java code. `http://www.kaffe.org/`, 1997.

[307] R. Wolski, N. T. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *J. of Future Generation Computing Systems* (to appear), 1999. `http://www.cs.ucsd.edu/users/rich/papers/nws-tr.ps.gz`. Also as TR CS98-599, Department of Computer Science, University of California at San Diego, September 1998.

[308] N. Yazici-Pekergin and J.-M. Vincent. Stochastic Bounds on Execution Times of Parallel Programs. *IEEE Trans. on Software Engineering*, 17(10):1005–1012, 1991.

[309] C. Yoshikawa, B. Chun, P. Eastham, A. Vahdat, T. Anderson, and D. Culler. Using Smart Clients To Build Scalable Services. In *Proc. USENIX Ann. Technical Conf.*, Anaheim, CA, January 1997. `http://now.CS.Berkeley.EDU/SmartClients/usenix97.ps`.

[310] J. W. Young. A First Order Approximation to the Optimum Checkpoint Interval. *Comm. of the ACM*, 17(9): 530–531, 1974.

[311] A. Yu, W. Cox. Java/DSM: A Platform for Heterogeneous Computing. In *Proc. Workshop on Java for Science and Engineering Computation*, pages 1213–1224, Las Vegas, NV, June 1997. `http://www.cs.rice.edu/~weimin/papers/java97.ps`.

[312] S. Zhou, X. Zheng, J. Wang, and P. Delisle. Utopia: a Load Sharing Facility for Large, Heterogenous Distributed Computer Systems. *Software: Practice And Experience*, 23(12):1305–1336, 1993.

[313] A. Ziv and J. Bruck. An On-Line Algorithm for Checkpoint Placement. *IEEE Trans. on Computers*, 46(9): 976–985, 1997.

Authorindex

Lebenslauf

Addresse:	Fritz Holger Karl Gossowstr. 4 10777 Berlin
Geburtsdatum:	15. Feburar 1970 in Eberbach
Staatsangehörigkeit:	Deutsch

Ausbildung

April 1989	Abitur am Hohenstaufen-Gymnasium Eberbach Note: 1,0
Oktober 1990 – Juli 1996	Studium der Informatik an der Universität Fridericiana, Karlsruhe Abschluß als Diplom-Informatiker Note: mit Auszeichnung
September 1993 – Mai 1994	Auslandsstudium in Computer Science an der University of Massachusetts, Amherst
seit August 1996	Stipendiat des Graduiertenkollegs “Kommunikationsbasierte Systeme” an der Humboldt-Universität zu Berlin
Januar 1997	Herman Billing Preis der Universität Fridericiana, Karlsruhe
August 1997 – April 1998	Forschungsaufenthalt an der New York University, Projekt “Metacomputing in Large Asynchronous Networks”

Berufserfahrung

Oktober 1992 – Juni 1995	Tutor und wissenschaftliche Hilfskraft an der Universität Friderciana, Karlsruhe
August 1995 – September 1995	Praktikant bei Siemens Karlsruhe

Berlin, den 9. Juni 1999

Vita

Address:	Fritz Holger Karl Gossowstr. 4 10777 Berlin
Date of birth:	February 15, 1970 in Eberbach
Citizenship:	German

Education

April 1989	Abitur at the Hohenstaufen-Gymnasium Eberbach GPA: 1,0
October 1990 – Juli 1996	Student of Informatik at Universität Fridericiana, Karlsruhe graduated as Diplom-Informatiker GPA: with honors
September 1993 – Mai 1994	Studies in computer science at University of Massachusetts, Amherst
since August 1996	Member of the Graduiertenkolleg “Kommunikationsbasierte Systeme” at Humboldt-Universität zu Berlin
Januar 1997	Herman Billing Award of the Universität Fridericiana, Karlsruhe
August 1997 – April 1998	Visiting researcher at New York University, Project “Metacomputing in Large Asynchronous Networks”

Professional Experience

October 1992 – June 1995	Teaching and Research Assistant at Universität Fridericiana, Karlsruhe
August 1995 – September 1995	Intern at Siemens Karlsruhe

Berlin, 9th June 1999